CONCURRENCY ABSTRACTIONS FOR PROGRAMMING LANGUAGES USING OPTIMISTIC PROTOCOLS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Adam Welc

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2006

Purdue University

West Lafayette, Indiana

To my parents.

ACKNOWLEDGMENTS

I would like to start with expressing my gratitude towards both of my co-advisors, Tony Hosking and Suresh Jagannathan. I really appreciate all the help, support and constant encouragement I received from them throughout all the years we spent working together. I would also like to thank Jan Vitek for serving on my committee and being very supportive of the research directions I decided to pursue. I am also grateful to T.N. Vijaykumar for agreeing to become a member of my committee.

During my years at Purdue I have made many friends who made the time I spent in the graduate school a lot more pleasant. To name just a few, Dennis Brylow, Joanne Lasrado, Piotr Osuch, Paul Ruth, Marta Zgagacz as well as both my labmates from the CS department and people from the "Polish group" in general. My special thanks to Natalia Nogiec and Phil McGachey for always being there for me both in good and bad times. I am also grateful to Adam Chelminski, Przemek Kopka, Justyna Reiska, Piotr Swistun and Krzysztof Waldowski who, despite staying in Poland while I moved to the US, remained very good friends that I could always go back to.

I thank my parents for helping and supporting me not only during my graduate school experience but also throughout all the years preceding it. Many thanks for all the encouragement also to my other family members, especially my grandparents.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Welc, Adam. Ph.D., Purdue University, May, 2006. Concurrency Abstractions for Programming Languages Using Optimistic Protocols. Major Professors: Antony Hosking and Suresh Jagannathan.

Concurrency control in modern programming languages is typically managed using mechanisms based on mutual exclusion, such as mutexes or monitors. All such mechanisms share similar properties that make construction of scalable and robust applications a non-trivial task. Implementation of user-defined protocols synchronizing concurrent shared data accesses requires programmers to make careful use of mutual-exclusion locks in order to avoid safety-related problems, such as deadlock or priority inversion. On the other hand, providing a required level of safety may lead to oversynchronization and, as a result, negatively affect the level of achievable concurrency.

Transactions are a concurrency control mechanism developed in the context of database systems. Transactions offer a higher level of abstraction than mutual exclusion which simplifies implementation of synchronization protocols. Additionally, in order to increase concurrency, transactions relax restrictions on the interleavings allowed between concurrent data access operations, without compromising safety.

This dissertation presents a new approach to managing concurrency in programming languages, drawing its inspiration from optimistic transactions. This alternative way of looking at concurrency management issues is an attempt to improve the current state-of-the-art both in terms of performance and with respect to software engineering benefits.

Three different approaches are presented here: revocable monitors are an attempt to improve traditional mutual exclusion, safe futures propose a new way of thinking about concurrency in a context of imperative programming languages and, finally, transactional

monitors try to reconcile transactions and mutual exclusion within a single concurrency abstraction.

1 INTRODUCTION

This thesis proposes a new way of looking at concurrency management in programming languages to allow both software engineering and performance improvements. Our approach draws its inspiration from optimistic transactions developed and used in the database community and constitutes an alternative to the more traditional way of providing concurrency control, namely mutual exclusion.

In this chapter we describe the most popular methods currently used to manage concurrency in both programming languages and databases. We also discuss the motivation behind our attempt to apply solutions drawing on optimistic transactions to a programming language context. At the end of the chapter we summarize our discussion in a thesis statement.

1.1 Concurrency Control for Programming Languages – Mutual Exclusion

Most modern programming languages, such as Java or C#, provide mechanisms that enable concurrent programming, where *threads* are the units of concurrent execution. Concurrency control in these languages is typically managed using mechanisms based on mutual exclusion to *synchronize* concurrent accesses of the shared resources (*e.g.*, memory) between multiple threads. In most cases synchronization mechanisms are used to protect regions of code designated by the programmer, containing operations accessing shared resources.

A *mutex* is the simplest example of such a mechanism. A thread wishing to execute the region of code protected by a mutex must first successfully *lock* the mutex. Only one thread is allowed to lock a mutex at any given time – this way exclusive access to the protected region of code is guaranteed. The mutex is *unlocked* when the thread exits the protected region. In other words, a mutex is essentially a simple mutual-exclusion

lock. C# and Modula-3 are examples of languages using mutexes for synchronization. A *semaphore* is a generalization of a mutex – it allows a fixed number of threads (determined upon semaphore creation) to execute within the protected region of code at the same time. Semaphores are most commonly used for synchronization at the operating system level.

Another popular synchronization mechanism is the *monitor*, originally proposed by Brinch-Hansen [26] and further developed by Hoare [31]. In its original interpretation, a monitor consists of the following elements: a set of routines implementing accesses to shared resources, a mutual-exclusion lock and a monitor invariant that defines correctness of the monitor's execution. Inclusion of the notion of correctness makes monitors a higher level mechanism compared to mutexes or semaphores. Monitors also support event signaling through *condition variables*. A thread executing a monitor's routine must acquire the mutual-exclusion lock before entering the routine - only one thread is allowed to execute within the same monitor at a given time. The lock is held until the thread exits the routine or until it decides to *wait* for some condition to become true using a condition variable (the waiting thread releases the lock). A thread causing the condition to become true can use the condition variable to *notify* the waiting thread about the occurrence of this event. The waiting thread can then re-acquire the monitor's lock and proceed.

The existing monitor implementations for Java and C# are modified with respect to this original interpretation. Each monitor is associated with an object and protects an arbitrary region of code designated by the programmer, called a *synchronized block*. A monitor still enforces mutually exclusive access to the code region but provides no additional guarantee with respect to correctness of execution within the monitor. Before a thread is allowed to execute the code region protected by a monitor, it must *acquire* the monitor. The monitor is *released* when execution of the protected region completes. Limited support for event signaling is supported – threads may wait on monitors and use them to notify other threads, but support for condition variables is missing. Additionally, monitors can be nested – after acquiring a monitor, a thread my acquire additional monitors without releasing the one it already holds, as well as re-enter the monitors it does hold.

T	T'

`void totalBalance() {`
`  synchronized (mon) {`
`  b1 = checking.getBalance();`
`  b2 = savings.getBalance();`
`  print(b1 + b2);`
`  }`
`}`
	`void transfer(int amount) {`
	`  synchronized (mon) {`
	`  checking.withdraw(amount);`
	`  savings.deposit(amount);`
	`  }`
	`}`

Figure 1.1. Bank account example

Synchronization mechanisms based on mutual exclusion and most commonly used in programming languages, that is mutexes and monitors, share similar properties. They are typically used to mediate concurrent accesses to data items residing in shared memory performed within the code regions they protect. Because only one thread is allowed to enter a protected region, it is guaranteed that accesses to shared data performed by this thread are *isolated* from accesses performed by the others. Also, all updates to shared data performed by a thread within a protected region become visible to other threads *atomically*, once the executing thread exits the region.

Enforcing such a strong restriction on the interleaving of concurrent operations is, however, not always necessary to guarantee isolation and atomicity. Consider the code fragment in Figure 1.1, using mutual exclusion monitor for synchronization. Thread T computes the total balance of both checking and savings accounts. Thread T' transfers money between these accounts. Operations of both threads are protected by the same monitor. The expected result of these two threads executing concurrently is that thread T' does not modify either account while thread T is computing the total balance – otherwise, the computed total might be incorrect. In other words, thread T is expected to observe the state of both accounts either before thread T' performs a transfer or after the transfer is completed. Using a mutual exclusion monitor for synchronization certainly guarantees exactly this behavior. Because only one thread is allowed to enter a region protected by the monitor at any given time, execution of threads T and T' may result only in two different

T	T'
rd(checking)	
rd(savings)	
	rd(checking)
	wt(checking)
	rd(savings)
	wt(savings)

(a)

T	T'
	rd(checking)
	wt(checking)
	rd(savings)
	wt(savings)
rd(checking)	
rd(savings)	

(b)

Figure 1.2. Serial executions

(serial) executions illustrated in Figure 1.2 . Figure 1.2(a) illustrates a sequence of data access operations when thread T executes all its operations before thread T' (withdrawal and deposit operations involve both a read and an update of the account balance). Figure 1.2(b) illustrates the opposite situation – a sequence of operations when thread T' executes all its operations before thread T.

We observe, however, that there exist other, more relaxed, interleavings of operations performed by threads T and T' that would result in the exact same (safe) behavior. Consider the execution illustrated in Figure 1.3. Its effects from the point of view of threads T and T' as well as with respect to the final result of the deposit operation are equivalent to the execution in Figure 1.2(a). Similar (safe) interleavings can be found under different scenarios, even when interaction among multiple concurrent threads is much more complicated, leading to a potentially significant increase in achievable concurrency.

T	T'
	rd(checking)
rd(checking)	
	wt(checking)
	rd(savings)
rd(savings)	
	wt(savings)

Figure 1.3. Interleaved executions

Unfortunately extracting additional available concurrency using mechanisms based on mutual exclusion is difficult. This is a direct consequence of trying to use a low level mechanism, such as mutual exclusion locks, to express higher level safety properties, such as isolation and atomicity. An attempt to achieve the desired level of performance may lead to under-synchronization, and consequently to violation of safety properties. Over-synchronization, on the other hand, may easily cause reduction in realizable concurrency and thus performance degradation.

Additionally, synchronization mechanisms based on mutual exclusion are not easily composable, especially if nesting is prohibited – consider the case when library code is synchronized, but details of the synchronization protocol are hidden from the library user. Allowing for these mechanisms to be nested aids composability, but may lead to other difficulties, such as *deadlock*. Deadlock occurs when threads waiting for other threads to release their mutual-exclusion locks form a cycle. Also, in a priority scheduling environment, *priority inversion* may result if a high-priority thread is blocked by a lower priority thread. These problems are exacerbated when building large-scale systems, where multiple programmers work on different parts of the system separately and yet are obliged to reconcile the low-level details of the synchronization protocol across different system modules.

These observations lead us to consider alternative concurrency control mechanisms, such as transactions, that help in alleviating problems related to using mutual exclusion.

1.2 Database Concurrency Control – Transactions

Traditionally, transactions have been used as a concurrency control mechanism in database systems [24]. A transaction is a fragment of an executing program that accesses a shared (persistent) database concurrently with other transactions. Transactional execution guarantees certain properties concerning these concurrent accesses, depending on a particular transaction model. We say that execution of a transaction is *safe* if it does not violate any of the transactional guarantees. The behavior of a transaction is controlled by the

following actions: *begin*, *commit* and *abort*. The execution of a transaction starts with the begin action followed by a sequence of data access operations. If it is determined that the execution of these operations does not violate any transactional guarantees, the transaction can execute the commit action (gets *committed*) and the effects of its execution become permanent with respect to the state of the shared database. If the transactional guarantees are violated, the transaction is aborted and all the effects of its execution (with respect to the shared state) are discarded.

Many transaction models have been developed over the years, reflecting different notions of safety. One of the most popular ones is the *ACID* model [24].

1.2.1 ACID Transactions

Execution of a transaction is safe according to the ACID model if it satisfies the following four properties:

- Atomicity – no *partial* results of a transaction become permanent with respect to the state of the database (an all-or-nothing approach),
- Consistency – execution of a transaction brings the database from one consistent state (with respect to internal database constraints) to another consistent state,
- Isolation – the operations of one transaction are isolated from the operations of all other transactions (*i.e.*, from a transaction's point of view it appears as if it is the only one executing in the system),
- Durability – the effects of a transaction must never be lost after it commits

The isolation property can be enforced by executing transactions serially. However, this may restrict available concurrency. Fortunately, unlike mutual exclusion, transactions do not enforce any particular interleaving between concurrently executing operations . It is safe to allow interleaved execution so long as the operations of the concurrent transactions are *serializable*. That is, it is sufficient if transactions produce the same results as if they execute serially.

All the existing protocols that enforce ACID properties can be generally divided into two major groups: *pessimistic* and *optimistic*.

1.2.2 Pessimistic Protocols

Pessimistic protocols assume that multiple concurrent transactions frequently compete for access to shared state. In order to prevent concurrent modifications of the shared state from violating serializability (and thus compromising isolation), pessimistic protocols typically lock the data elements they operate on. Because pessimistic protocols perform updates in-place (as opposed to delaying their propagation to the shared space), they must *log* enough information about the updates to be able to undo them in case of an abort. We call transactions supported through the use of pessimistic protocols *pessimistic transactions*.

One of the most popular locking protocols is *two-phase-locking* (or *2PL*) [24]. It divides a transaction into two phases: the growing phase when locks are only acquired and the shrinking phase when locks are only released. In its strictest, and most popular form (the non-strict version may lead to cascading aborts[1]), 2PL defers release of any of its locks until it terminates (commits or aborts). The 2PL protocol distinguishes two types of locks: shared locks acquired before a data element is read, and exclusive locks acquired before a data element is written. A data element may be locked by multiple transactions in the shared mode (we say that shared locks are mutually *compatible*) but only by one transaction in the exclusive mode (we say that an exclusive lock is in *conflict* with any other lock). A transaction is blocked when trying to acquire a conflicting lock – it is allowed to proceed only once the conflicting lock is released. Unfortunately, 2PL (and most other locking protocols) can result in *deadlock*. A deadlock occurs when two (or more) transactions wait for each other's (conflicting) locks to be released forming a cycle – it can be resolved by aborting one of the transactions involved. Some form of deadlock detection (or prevention) protocol must therefore also be deployed in a system using 2PL.

[1] All transactions that have seen updates of a transaction being aborted must be aborted as well.

1.2.3 Optimistic Protocols

The assumption underlying optimistic protocols is that the amount of sharing with respect to data elements accessed by concurrent transactions is low. Therefore transactions are allowed to proceed with their updates until termination in the hope that no violations of serializability ever occur. This optimistic assumption must however be validated upon transaction completion – if it holds, the transaction is committed, otherwise it is aborted and re-executed. We call transactions supported through the use of optimistic protocols *optimistic transactions*.

Optimistic transactions have been originally proposed by Kung and Robinson [37]. The execution of a transaction is divided into three phases: a *read phase*, a *validation phase* and a *write phase*. In the read phase transactional operations are redirected to a local log instead of operating directly on shared data. This way premature exposure of the transaction's computational effects is avoided (allowing transactions to update shared data in-place could lead to cascading aborts). The validation phase is responsible for detecting potential serializability violations. If a transaction successfully passes the validation test, all transactional updates are propagated to the shared space in the write phase and the transaction commits. Otherwise all updates are discarded, the transaction aborts, and is re-executed.

1.3 Motivation

Synchronization protocols based on mutual exclusion have several deficiencies as described in Section 1.1. Recognition of this fact has prompted us to consider transactions as an alternative way to manage concurrency in programming languages.

The application of transactions in the context of a programming language poses new challenges that are quite different to that of using transactions in a database environment. Issues related to management of database (persistent) state, such as durability and consistency in the ACID model, become irrelevant. Instead, transactions manage concurrency and preserve safety properties with respect to the *volatile* shared heap, whose contents do

not survive system's shutdown or failure. Thus, the set of properties of the ACID model that need to be preserved becomes limited to atomicity and isolation.

Since transactions are a much higher level construct, they have potential for mitigating the mismatch currently existing between reasoning about properties of concurrent programs at a high level and implementing protocols enforcing these properties at a considerably lower level. Thus, the software engineering benefits from using transactions may be significant. Additionally, because transactions allow more relaxed interleavings of concurrent operations, and so potentially enable a higher degree of concurrency than solutions based on mutual exclusion, they may also lead to improved performance of concurrent applications.

At the same time, synchronization mechanisms based on mutual exclusion are unlikely to disappear any time soon. One of their main advantages is that they can be very efficient if contention on access to regions they protect is low. On the other hand, the effectiveness of transactional mechanisms is proportional to the amount of data shared among concurrent transactions. In the case of pessimistic transactions, data items are locked to prevent concurrent access. This way, if the amount of data sharing is significant, the achievable concurrency may be significantly reduced. Additionally, deadlocks may occur more frequently and yet the cost of maintaining transactional properties (*e.g.*, related to locking of data items) still needs to be paid. In the case of optimistic transactions, excessive data sharing may result in the increased number of aborts, yielding a similarly negative effect.

Therefore, our intention is to use transactions to manage concurrency only when beneficial, such as when the amount of data sharing is low, rather than uniformly replacing mechanisms based on mutual exclusion. We still have to ensure that transactions are extremely light-weight in order to remain competitive with existing solutions for managing concurrency. We believe that optimistic transactions fulfill these requirements better than pessimistic ones. When using pessimistic transactions, additional mechanisms are required to avoid cascading aborts or deadlocks in case a locking protocol is used, while still preserving the requirement to support logging. Also, the cost of per-data-item locking, required in this case, tends to be significant.

1.4 Thesis Statement

Optimistic transactions represent a feasible alternative to a traditional approach to managing concurrency in programming languages based on mutual exclusion. Solutions utilizing optimistic transactions can be not only beneficial from a software engineering point of view but can also lead to significant performance improvements.

1.5 Thesis Overview

In Chapter 2 we discuss several mechanisms required to support optimistic transactions. Chapter 3 contains discussion of the related work. In the subsequent three chapters we describe our own approaches to solving problems related to writing concurrent applications in Java, using optimistic transactions as a foundation. In Chapter 4 we discuss how traditional Java monitors can be augmented using transactional machinery to alleviate problems related to priority inversion and deadlock. In Chapter 5 we examine how optimistic transactions can be applied to support the *futures* abstraction in Java. In Chapter 6 we describe how mutual exclusion and optimistic transactions can co-exist within a single framework. Finally, Chapter 7 contains conclusions and discussion of the future work.

2 SUPPORT FOR OPTIMISTIC TRANSACTIONS

The task of providing support for optimistic transactions is in our case set in the context of an existing programming language environment, supporting its own set of programming language related features (*e.g.*, memory management, exceptions *etc.*). This makes the design of the transactional support quite different from when it can be build from ground up, which is the case in the database world. We may sometimes modify and re-use prior mechanisms, but in general it is a non-trivial task to superimpose transactions over these mechanisms and guarantee their seamless integration.

2.1 Design Goals

One of our main design goals for a system offering optimistic transactions as a concurrency control mechanism in a programming language context is *programmer-friendliness*. A typical programmer already has some level of experience in using traditional approaches of managing concurrency that are usually based on mutual exclusion (*e.g.*, mutexes, monitors or semaphores). It is unlikely that programmers will be willing to abandon all their (potentially considerable) expertise in using these mechanisms in favor of a completely new approach they must learn from scratch.

Therefore we opt for *simplicity* in our design. If new language abstractions need to be introduced, they should be few and their properties easy to understand. Wherever possible we strive for partial or full *transparency* – the exposure of transactional machinery to the programmer should be minimal.

At the same time, our approach must be *general* enough to be usable in practice. Since we introduce transactions in the context of an already existing language, a considerable amount of legacy code is likely to exist. Our solution should therefore be at least partially backward-compatible (*e.g.*, to allow re-use of existing library code). Additionally, source

code may not always be available – its absence should not preclude using transactions for managing concurrency within legacy code.

Some of these design goals, such as programmer-friendliness or simplicity, influence high-level aspects of the system, such as the form in which transactions are exposed to the programmer. We address these issues when discussing specific solutions in subsequent chapters. The other goals, such as transparency and generality, must be taken into account at a much lower level, such as when considering design choices for foundational mechanisms required to support optimistic transactions.

Several such mechanisms are required to enable use of optimistic transactions in a programming language context. Their equivalents exist in the world of traditional database system, but their adaptation to a programming language context requires careful consideration of various design and implementation trade-offs. In particular, design choices proven to be effective in the context of database systems may not necessarily be equally applicable to a programming language environment.

We distinguish three types of such foundational mechanism:

- Logging – a mechanism used to record (in a log) transactional operations accessing the elements of shared data. Depending on the specifics of the transaction semantics, a log may serve two purposes. Transactional operations may be redirected to the log and applied to the shared space upon commit of the transaction. Alternatively, if transactional updates are performed in-place, information recorded in the log may be used to revert their effects upon abort of the transaction.

- Dependency tracking – a mechanism used to detect violations of atomicity and isolation. Multiple transactions executing concurrently may access the same data items in the shared space, creating dependencies among data access operations. Dependency tracking is responsible for detection of all dependencies that lead to violations of transactional properties. All transactions violating these properties are aborted.

- Revocation – a mechanism supporting the abort operation. Conceptually, revocation consists of two parts: first, all the effects of transactional execution (both with

respect to shared and local state) must be reverted and, second, control must be returned to the starting point of the aborted transaction (to enable re-execution).

Detailed descriptions of these mechanism are given below.

2.2 Logging

Traditionally [24], in the context of (persistent) database systems, logging is used for transaction recovery. A log is an entity logically[1] separate from the actual persistent store and contains all the information necessary to bring the persistent store to a consistent state in case of unexpected events. These include system failures or explicit (triggered by the user) as well as implicit (*e.g.*, initiated to resolve deadlock) transaction aborts. It is assumed that the effects of updates performed by transactions do not have to be immediately propagated to the persistent store, whether for performance reasons or to satisfy requirements of a particular transaction model. It is sufficient that the log contains all the information about the updates necessary to enforce the transactional (*e.g.*, ACID) properties and possesses the ability to survive system failures.

In case of failures, effects of operations performed by committed transactions should not be lost, in order to satisfy the durability property. At the same time partial effects produced by transactions that have not yet committed should not become permanent because of the atomicity requirement. Information about the transactional updates recorded in the log can thus be used to *undo* the effects of uncommitted transactions and *redo* operations of the committed ones. Similarly, effects of a transaction being aborted can be undone using information from the log.

Two major groups of logging protocols exist: *physical logging* and *logical logging*. Physical logging is typically realized by recording both a *before image* and *after image* of a data element taken before and after performing an update, respectively. This greatly simplifies implementation of undo and redo operations – the only action required is to retrieve the value from the log and apply it to the appropriate data element. However,

[1] The log may itself reside in persistent storage, if not in the application store.

since database update requests tend to be declarative and may concern a large number of data elements, physical logging may incur significant memory overhead when recording all the requested updates. For example, a request to update a large table by incrementing the value of each element stored in the table would most likely incur generation of a large number of log records. When logical logging is used, the same request can be very succinctly represented in the log by recording the request itself and the accompanying parameters. Therefore, logical logging is considered to be a better solution for logging of updates in traditional database systems [24].

2.2.1 Volatility

The application of transactions to a programming language context changes the way logging is used. The notion of persistent store is no longer present. The updates performed by transactions are reflected only in the volatile store (*i.e.*, in the shared heap) and issues related to maintaining persistent state become irrelevant. Thus, the log itself can be volatile which greatly simplifies log management because there is no need for the log to survive a system failure. Even though failure recovery is no longer present, logging must still support redo or undo operations, depending on the transaction model. If a transaction directly updates data in the shared store, the log is used to undo the effects of aborted transactions. Otherwise, the log is used to redo updates of committing transactions to propagate their effects to the shared store.

Logical logging loses its advantage over physical logging in a programming language context, since shared heap operations only access one memory word at a time. We therefore choose to use physical logging, which in this context seems to be the simplest and the least expensive solution. Two methods of realizing physical logging can be identified: one using a *sequential* log to record all updates to shared data performed within a transaction, and the other using per-transaction copies (so-called *shadow copies*) of shared data elements to record updates to these elements. A sequential log records the effects of transactional operations in the order they occur. When shadow copies are used, information

about all updates to a given element performed by a transaction is represented by a single shadow copy.

In its purest form (described in Section 1.2.3), an optimistic transaction does not directly update shared data elements. This avoids premature exposure of updates in case of an abort. As a result, after performing a write, every subsequent read of the same element must consult the log for the most up-to-date value. If sequential logging is used, a read operation might involve scanning of the sequential log, potentially to its very beginning. Considering the pervasiveness of reads in modern programming languages, this could incur considerable run-time overhead. We believe that shadow copying is a preferred solution in this case. However, if premature exposure of updates is prevented (*e.g.*, by some separate mechanism) and a transaction is allowed to operate directly on the shared data, no scanning of the log is required while the transaction is running. Using a sequential log might be a better solution in this situation. We use sequential logs in our implementation of revocable monitors described in Chapter 4, where mutual exclusion is used to prevent premature exposure of updates.

Shadow copying is essentially a form of shared data *versioning*. Multiple versions of the same data element, created by different transactions, may exist at the same time. We use versioning to implement logging in the case of safe futures (described in Chapter 5) and transactional monitors (described in Chapter 6). For the following discussion concerning the versioning mechanism we assume that transactions operate on versions (instead of operating directly on the shared data) and propagate updates to the shared heap at the time of commit.

2.2.2 Versioning

A transaction needs to be able to access versions it has created. One obvious approach is to keep versions created by a given transaction in some data structure maintained "on the side" and accessible by this transaction. Since the association between a version and the original data element must be maintained, a hash-table seems to be a natural choice for

such a structure. However, the cost of performing a hash-table operation at all transactional reads and writes would be overwhelming (especially considering the unpredictability of operations concerning hash-table maintenance, such as resizing, re-hashing, *etc.*). Also, the size of the hash-table (and thus, when considering chaining in the hash-table, the time to access a version) becomes directly proportional to the number of data elements accessed by a transaction. It would seem that in the case of optimistic transactions a scheme where time to access a version is proportional to the amount of data sharing between transactions would be more desirable. Therefore we choose to keep versions on lists directly associated with shared data elements. Accessing a version involves searching a list, which is expected to be short when the amount of data sharing among different transactions is small (which is one of the assumptions motivating use of optimistic transactions).

At the time of commit, a transaction must be able to propagate information about updates from the versions it created to the data elements in the shared heap. Application of updates may be done *eagerly* and simply involve copying the new values from a version to the original data element. This, however, means that copying for every updated element of shared data is performed twice, once when the version is created, and a second time when updates are propagated to the shared store. Additionally, if an element of shared data modified within the scope of a transaction is never accessed again, eager application of updates becomes a source of unnecessary overhead. We adopt a different solution and propagate updates *lazily*. The association between the original data element and its version is maintained beyond the point of transaction commit. At the time of the commit, the version created by the committing transaction is designated as the one containing most up-to-date values and used for all subsequent accesses. As a result, all subsequent accesses (including the non-transactional ones) must be redirected to access this version.

2.3 Dependency Tracking

In general, unless an external mechanism (*e.g.*, mutual exclusion in the implementation of revocable monitors described in Chapter 4) guarantees otherwise, the operations of

multiple concurrent transactions can be arbitrarily interleaved. However, in order to satisfy the isolation requirement, the final effects of concurrent execution must be serializable, Some form of a data dependency tracking mechanism is therefore required to validate serializability of transactional operations.

One of the important trade-offs that should be considered when choosing the most appropriate dependency tracking mechanism is that between precision and incurred run-time overheads. Conservative (imprecise) solutions are typically less expensive at run-time but may lead to detection of spurious (non-existing) dependencies, which might lead to an increased number of serializability violations being detected. Precise solutions detect serializability violations only in situations when they really occur, but their run-time cost may be prohibitive.

Precise solutions typically rely on the ability to record information about all heap locations accessed by a transaction. In order to validate if operations of a transaction are serializable, all the heap locations accessed by the transaction are inspected to verify if they have been accessed by other concurrently executing transactions. The cost of the validation procedure in this case is quite significant – additional information must be associated with every heap location and, as a result, the number of shared data accesses performed by the transaction may be significantly increased. In the worst case the number of accesses is doubled since every regular transactional access can be followed by another access during the validation phase.

In a system using optimistic transactions, however, it is assumed that the number of concurrent accesses to a given data element (and thus the number of dependencies that might lead to serializability violations) is low. Therefore, detection of spurious dependencies by the mechanism chosen for data dependency tracking should not dramatically increase the number of serializability violations detected. We believe that the cost of performing an unnecessary revocation, on the rare occasion a spurious dependency is detected, is going to be outweighed by the low run-time costs associated with a conservative approach.

We choose to record data accesses in a fixed-size table. The conservatism of the approach manifests itself in the fact that the same table entry may represent accesses to different data items. Only one bit of information is used to record access to a given shared data element – it is set after the first access to a given element. The table thus becomes essentially a bit-map. We distinguish two types of maps, a *read map* (to record reads) and *write map* (to record updates). Non-empty intersection of maps containing accesses from different transactions indicates existence of dependencies between operations of these transactions. Mechanisms relying on the notion of read and write maps to track data dependencies are used in the case of safe futures (described in Chapter 5) and transactional monitors (described in Chapter 6).

2.4 Access Barriers

Our desire to preserve transparency dictates that the exposure of both logging and dependency tracking mechanisms to the programmer should be minimal. Therefore we discard solutions where the programmer is asked to designate specific elements of shared data to be amenable for transactional concurrency control or is forced to explicitly distinguish transactional data accesses from the non-transactional ones. This would not only violate our transparency requirement, but also hinder generality of our approach. A programmer wishing to use transactions to mediate shared data accesses within the system libraries would have to gain access to their source code and modify it, which is often difficult and sometimes even impossible.

Instead, we support logging and data dependency tracking mechanisms through transparently (hidden from the programmer and independent of the type of shared data element) augmented versions of all shared data access operations. These *access barriers* (or simply barriers) originate in the area of automatic memory management, that is garbage collection [32]. In this context, the barriers are used to monitor operations performed by the application (called a *mutator*) to access data items residing in a shared heap. Two types of barriers exist: *read barriers* encapsulating actions to be executed when the mutator reads

a reference from the heap and *write barriers* encapsulating actions to be executed when it writes a reference to the heap. Typically, only one type of barrier is used at a time, depending on the specific garbage collection algorithm. The barriers can be used to partition the heap into regions that can be collected separately for improved performance or to reconcile actions of the mutator and the garbage collector in case they execute concurrently.

We generalize the notion of garbage collection barriers in order to provide support for transactional accesses to the shared heap. In order to support logging of shared data accesses, we use barriers to augment all operations on the shared data items (including reads and writes of primitive values, not not only reference loads and stores). In order to correctly track dependencies between operations accessing the heap, both reads and writes may have to be taken into account and thus read and write barriers can be used simultaneously.

Barriers are usually provided as code snippets implementing the augmented data access operations and are inserted by the compiler. Insertion of barriers at the source code level is infeasible because source code may not always be available. We assume that an optimizing compiler is going to be used at some stage of the compilation process and advocate for barrier insertion by the optimizing compiler. This way existing compiler optimizations, such as escape analysis, may be used to reduce barrier-related overheads.

2.5 Revocation

A transaction that has been determined to violate the transactional properties is aborted. The effects of operations performed by the transaction must be undone and the transaction must be re-executed. The details of the revocation procedure should be kept hidden from the programmer, because of our transparency requirement. Ideally, a programmer should not even be aware that revocations take place in the system – the final effect of executing a transaction that at some point gets aborted should be as if this transaction had never started executing its operations in the first place.

The procedure for revoking a transaction consists of several steps. If a transaction operates directly on shared data, all its updates must be undone (by using information from the log – if a transaction does not modify shared data, no action is required here) and the control must be returned to the point where the transaction started executing. Additionally, all the local state modified by the transaction (*e.g.*, local variables) must be reverted to reflect the situation before the transaction began.

In the case of traditional database systems, the revocation procedure is an inherent part of the database engine. A transaction is the smallest unit of concurrent execution and fully encapsulates all the operations whose effects need to be undone. As a result, a mechanism to revoke a transaction can be directly embedded in the database engine. When transactions are used in a programming language, they are typically superimposed over language-specific concurrency mechanisms (such as threads), which may complicate the revocation procedure.

One of the challenges we have to face when reconciling transactions with threads is transaction re-execution. If a transaction can be easily encapsulated into an executable unit (*e.g.*, function, method or procedure), returning control to the point where the transaction started executing is trivial. The revocation procedure may simply re-execute the unit after invoking a routine responsible for restoration of both the local state and the shared state (if necessary). In general, however, this level of encapsulation may not be available – a transaction may simply be designated as a sequence of operations performed by a thread (which may not even be lexically scoped). In this case, a more complicated mechanism to support revocation is required.

Fortunately, in most modern languages there already exists a mechanism to allow ad-hoc modifications to the control-flow during the execution of a program – *exceptions*. We take advantage of the existence of this mechanism. We wrap the block of code representing a transaction within an exception scope that catches a special `Revoke` exception. Revocation is triggered internally (at the level of the language's run-time system) by throwing the `Revoke` exception. The exception handler catching this exception is then responsible for restoring the local state (and shared state if necessary) and returning control to the

beginning of the block of code representing the transaction. The local state from the point before the transaction begins is recorded in a data structure associated with the transaction. A routine responsible for recording local state and the exception handler may be inserted at any point during program compilation, but below the level of source code because of our design requirement for generality. Additionally, we must make sure that during the handling of the `Revoke` exception, no default handlers are executed. If this was not prevented, the transparency of the re-execution mechanism could be compromised. This style of re-execution procedure is used for revocable monitors (described in Chapter 4), safe futures (described in Chapter 5) and transactional monitors (described in Chapter 6).

Another difficulty in supporting revocations in a programming language is that the effects of some operations executed by a transaction, such as I/O, cannot be undone. Also, the behavior of some language-specific mechanisms, such as thread notification, may be affected by revocations. The situation is additionally complicated by our requirement to keep revocations hidden from the programmer. For example, multiple re-executions could cause multiple unintended thread notifications. We defer discussion of how these issues are handled to the subsequent chapters since the choice of specific techniques is dependent on the functionality provided by the system.

2.6 Transactions in Java

We realize our support for optimistic transactions in the context of Java, currently one of the most popular mainstream programming languages. We do not, however, see any major obstacles preventing application of the techniques we describe to other programming languages, such as C#. The choice of Java was driven mainly by its popularity and by the availability of a high-quality implementation platform, namely IBM's Jikes Research Virtual Machine (RVM) [4]. The Jikes RVM is a state-of-the-art Java virtual machine with performance comparable to many production virtual machines. It is itself written almost entirely in Java and is self-hosted (*i.e.*, it does not require another virtual machine to run). Java bytecodes in the Jikes RVM are compiled directly to machine code. The Jikes RVM's

distribution includes both a “baseline” and optimizing compiler. The “baseline” compiler performs a straightforward expansion of each individual bytecode into a corresponding sequence of assembly instructions. Our implementations target the Intel x86 architecture.

3 RELATED WORK

Difficulties in using mutual exclusion as a concurrency control mechanism have inspired several research efforts aimed at exploring the applicability of transactions as a synchronization mechanism for programming languages. The purpose of this chapter is to put our own effort of developing transactions-based techniques for managing Java concurrency in the context of other similar attempts. We describe a range of solutions centered around the concept of *software transactional memory* (*STM*) – an abstract layer providing access to transactional primitives (such as starting and committing transactions and performing transactional data access) from the programming language level. Broadly speaking, our own solutions fall into the same category. Our presentation covers solutions ranging from the very first implementations of STM to more recent sophisticated high-performance systems.

Shavit and Touitou [53] describe the first implementation of software transactional memory for multiprocessor machines – one transaction per processor can be executed at a time. Their approach supports *static transactions*, that is transactions that access a prespecified (at the start of a transaction) set of locations. They implement an STM of a fixed size (*i.e.*, a fixed number of memory locations) using two main data structures: a vector of cells containing values stored in the transactional memory and a vector describing the ownership of transactional memory cells. Additionally, every processor maintains a *transaction record* used to store information about its currently executing transaction, such as the set of all the cells its transaction is going to access. The execution of a transaction consists of three steps. First, a transaction attempts to acquire ownership of all the cells specified in the transaction record. Then, if ownership acquisition is successful, it computes the new values, stores the old values into the transaction record (to be returned upon successful commit) and updates the appropriate cells with the new values. Finally,

it releases ownership of the cells and commits. Inability to acquire ownership of the cells specified in the transaction record results in an abort.

Because of the requirement to acquire ownership of all the cells a transaction needs to access, transactions in Shavit and Touitou's system can be considered pessimistic. The need to revoke the aborted transactions does not exist here since no transactional operations are performed before ownership of all the required cells is acquired. In other words, if transactional operations are allowed to proceed, they will always complete successfully. Shavit and Touitou present a performance evaluation of their system based on simulation. Their conclusion is that concurrent lock-free data structures implemented using their STM would perform better than the same data structures implemented through manual conversion from their sequential counterparts.

A more general version of software transactional memory, dynamic STM, was developed by Herlihy *et al.* [30]. They built an implementation supporting both Java and C++. In their system, the requirement to pre-specify the locations that are accessed by a transaction is lifted. Their programming model is based on the notion of explicit transactional objects. Transactional objects are wrappers for regular Java or C++ objects and only accesses to transactional objects are controlled by the transactional machinery. Their system uses a version of pessimistic transactions with explicit locking – before a transactional object can be accessed within a transaction, it must be locked in the appropriate (read or write) mode. A locking operation on the transactional object returns a version (*i.e.*, a copy) of the encapsulated regular Java or C++ object, which is used by the transaction for all subsequent accesses. Every locking operation involves execution of the validation procedure to verify that no other transaction locked the same object in a conflicting mode (a conflict is understood in the same way as in the description of the 2PL protocol in Section 1.2.2). If another transaction holds a lock in the conflicting mode, user-defined contention managers are used to determine which of the two conflicting transactions should be aborted. As a result, a transaction may be aborted at an arbitrary point (aborts are signaled by throwing a run-time exception). Object versions created by an aborting transaction are automatically discarded, but it is the programmer's responsibility to decide whether the transaction

should be re-executed, and to implement this operation explicitly if needed. To validate the usefulness of their approach, Herlihy *et al.* implement several transactional versions of an integer set, varying the type of underlying data structure and experimenting with different contention managers. They demonstrate that their transactional implementations outperform an implementation of an integer set that uses coarse-grained mutual exclusion locks for synchronization.

An even more general proposal for the design and implementation of an STM has been recently[1] proposed by Harris and Fraser [27]. Their approach does not require objects to be specially designated to enable transactional access. Their solution is set in the context of Java. They use STM support to provide programmers with a new language construct, called `atomic`. The `atomic` keyword is used to designate a group of thread operations (in the form of a code block or a method) that are supposed to execute in isolation from operations of all other threads. The STM is responsible for dynamically enforcing that the execution of an atomic block or an atomic method indeed satisfies this property. The execution of general-purpose native methods (*e.g.*, supporting I/O) as well as Java's wait and notify operations is forbidden within atomic methods and blocks. Such situations are detected at run-time and signaled to the programmer by throwing an exception.

Harris and Fraser's approach uses optimistic transactions. Several data structures support transactional accesses. *Transaction descriptors* maintain information about currently executing transactions, such as transaction status and a list of heap accesses performed by this transaction. A transactional heap access is recored in the form of a *transaction entry*, and contains the old and the new value for the given location (updates are propagated to main memory only upon commit) as well as version numbers for those values (every time a new value is assigned to a location, the version number gets incremented). An *ownership function* maps heap locations to appropriate *ownership records*. Each ownership record holds the version number or transaction descriptor for its location (describing the ownership record's current owner). A version number indicates that some transaction has just committed and propagated its update to the heap; a transaction descriptor indicates a

[1]The first version of their STM was (independently) developed at about the same time as our own first prototype implementation of the system supporting optimistic transactions.

transaction that is still in progress. Ownership records record the history of transactional accesses and are used during commit to validate transactional properties and propagate updates to the heap. At commit time, all the required ownership records are acquired (locked), version numbers are used to verify the correctness of heap accesses (with respect to transactional properties), updates performed by the transaction are propagated to the heap and the ownership records are released (unlocked). If acquisition of ownership records fails (*i.e.*, one of the ownership records is already held by a different transaction) or if transactional properties have been violated, the transaction is aborted. Because an abort can only happen upon transaction completion, the revocation procedure is simple. Bytecode rewriting is used to encapsulate every group of atomic actions into a method that can simply be re-executed after all the information about updates performed by the aborting transaction is discarded. Harris and Fraser evaluate the performance of their system using several microbenchmarks, demonstrating the scalability of their STM implementation. The overall performance of the microbenchmarks implemented using their STM is competitive with that of the same microbenchmarks implemented using mutual exclusion.

An implementation of STM can be further refined using *revocable locks*, a lower-level optimistic concurrency mechanism introduced by Harris and Fraser [28]. Revocable locks are a general-purpose mechanism for building non-blocking algorithms. They have been designed to provide a middle-ground between using mutual exclusion and attempting to build non-blocking algorithms without any forms of lock (*e.g.*, using only atomic compare-and-swap operations). A revocable lock is associated with a single heap location and provides operations to access that location as well as operations to lock and unlock the location. A revocable lock can be held by only one thread at any given time. However, any thread attempting to acquire some lock already held by another thread always succeeds – the holder's ownership of the lock is revoked and its execution is displaced to the recovery function supplied with its own lock acquisition operation. In other words, after acquisition the lock is held until it is explicitly released by the holder or until its ownership is revoked by another thread.

Revocable locks have been used, as one of the case studies, to streamline the commit operation in Harris and Fraser's STM described above. A committing transaction acquires a revocable lock on its transaction descriptor. If a committing transaction tries to use an ownership record already used by a different (committing) transaction, it revokes the lock of the current ownership record's user and attempts to complete the remaining operations of the current user's commit procedure (and then re-try its own commit). This guarantees that only one thread at a time performs the operations of any given commit procedure – transaction descriptors are then in effect used to represent pieces of computation that different threads may wish to perform. As a result, a committing transaction attempting to use an ownership record already used by a different transaction does not need to be immediately aborted.

Harris *et al.* [29] explore the expressiveness and composability of software transactions in a port of Harris and Fraser's STM to Concurrent Haskell [33]. Concurrent Haskell is a functional[2] programming language which, compared to Java, opens new possibilities and different trade-offs for higher-level design decision. However, the implementation of the lower-level STM primitives for Concurrent Haskell is in principle similar to their implementation for Java – both systems use a similar flavor of optimistic transactions.

The basic concurrency control construct provided to Concurrent Haskell programmers is similar to the one available in the Java-based system – the atomic block. However, two additional constructs have been added to improve the expressiveness and composability of the transactions-based concurrency control machinery. The first one is a `retry` function, used within an atomic block to provide a way for the thread executing the block to wait for events caused by other threads. This function is meant to be used in conjunction with a conditional check of the value of some transactional variable. If the transactional variable has the expected value, the thread is allowed to proceed, otherwise its transaction is aborted and re-executed. The re-execution, however, does not start (*i.e.*, the thread is blocked) until at least one transactional variable previously used by the thread gets modified. Otherwise there would be no chance for the conditional check to yield a different

[2]Some operations may, however, produce side-effects.

result. The second construct is a `orElse` function whose role is similar to the select function used in operating systems. The `orElse` function takes two transactions as arguments. The function starts with an attempt to execute the first transaction. If the first transaction is retried then it is aborted and the `orElse` function attempts to execute the second transaction. If the second transaction is also retried then it is aborted as well and the execution of the whole `orElse` function is retried. The re-execution is postponed until at least one of the transactional variables used by either of the transactions passed as arguments is modified.

The STM implementation for Concurrent Haskell relies on the notion of explicit transactional variables. In other words, transactional guarantees are enforced only with respect to variables of a special (transactional) type. As a result, it can be statically enforced that transactional variables are manipulated only within atomic blocks. Another interesting feature of Concurrent Haskell's type system is that I/O operations can be distinguished from regular operations based on the static types of values they manipulate. This allows the implementation of STM to guarantee statically that no I/O operations are ever executed within atomic blocks. A detailed performance evaluation of the STM implementation is currently not available, since Concurrent Haskell is implemented only for uni-processors, but the preliminary results seem to be encouraging.

The most recent, high-performance implementation of STM has been proposed by Saha *et al.* [51]. Their focus is on exploration of different implementation trade-offs with respect to their effect on STM's performance. Their system provides both general-purpose transactional memory primitives (starting and committing transactions, transactional data accesses, *etc.*) and a transactional implementation of the multi-word atomic compare-and-swap operation. Their implementation is built on top of an experimental multicore run-time system (designed for future multicore architectures) supporting different programming languages, such as Java or C++.

Saha *et al.* use pessimistic transactions with a sequential log to record transactional updates. Their system supports two different levels of locking granularity: locking at the object level and locking at the level of cache lines, which at the same time determines the

level at which conflicting data accesses are detected. Locking at the object level is used only for small objects and locking at the level of cache lines in all other cases. Saha *et al.* experiment with two different types of locking protocols. The first one is essentially equivalent to the 2PL protocol described in Section 1.2.2, where data items are locked in either read or write mode before being accessed. The second protocol locks data items only before performing writes. The validity of reads is verified at commit time using version numbers similarly to the technique used in Harris and Fraser's STM described above. They experimentally determine that the performance of the second protocol is significantly better than that of the first one. Both locking protocols can lead to deadlock which is detected using time-outs. They also explore two ways of handling transactional updates. The first one buffers updates in a log and applies them to the shared heap at commit time. The second one performs updates in-place – information in the log is used to undo the updates in the case of abort. In their system the second approach yields better performance which is the direct result of using the sequential log to buffer updates. A transactional read following an update to the same location performed within the same transaction must observe the effect of the update, and the operation of retrieving this value from the sequential log is expensive. The overall performance of their system, as demonstrated using a set of microbenchmarks as well as a modified version of the real-life sendmail application, is comparable to or better than when mutual exclusion is used as a synchronization mechanism.

Daynès and Czajkowski [15] propose to use transactions in a slightly different context, that is as protection domains for applications running within the same address space. In their approach, every program executes as a transaction and every object is owned by a single transaction, which is responsible for authorizing access to this object. Responsibilities of transactions in their system, in addition to managing concurrency, include fault containment (incorrect behavior of one application should not affect the behavior of the others) and memory access control (access to certain regions of memory by an un-trusted application may be restricted). The use of transactions also facilitates safe termination of

applications – since every program executes as a transaction, its execution may be aborted at an arbitrary point and all its effects can be safely undone.

Their implementation, extending the Java HotSpot virtual machine version 1.3.1, is based on a pessimistic transaction model (described in Section 1.2.2) – items of shared state must be locked before they can be accessed. Transactions operate directly on the shared memory and a physical log associated with each transaction is used for the undo operation (upon abort of the transaction). The novelty of their approach is related to sharing of the lock state. Traditionally, there exists a one-to-one mapping between a locked resource (in this case – an object or an array in the main memory) and a data structure representing the state (mode) of a lock protecting this resource. Lock state sharing, implemented by by Daynès and Czajkowski, is inspired by an observation that the total number of distinct lock values in the system is typically small with respect to the number of the locked resources, that is many objects may be locked by two (or more) transactions in the same mode at the same time. A data structure representing the lock state consists of two bit-maps, one for read (shared) locks and one for write (exclusive) locks. This data structure is pointed to by an object's or array's header. Every slot in a bitmap represents a currently active transaction – if it is set, it indicates that a given transaction holds a lock on a given object or array in the mode specified by the type of the bit-map . This way of implementing data structures representing the lock state not only brings significant memory savings, but also enables efficient implementation of lock manager's operations, such as lock ownership tests. The overheads related to using transactions as protections domains reported by Daynès and Czajkowski are on the order of 25%.

4 REVOCABLE MONITORS

Difficulties arising in the use of mutual exclusion synchronization in languages like Java, such as *priority inversion*, have been discussed in Section 1.1. Since Java supports priority scheduling of threads, priority inversion may occur when a low-priority thread T_l holds a monitor required by some high-priority thread T_h, forcing T_h to wait until T_l releases the monitor. An example of a situation when priority inversion can occur is illustrated by the fragment of a Java program in Figure 4.1. Thread T_l may be the first to enter a given synchronized block (acquiring monitor `mon`) and block thread T_h while executing some (arbitrary) sequence of code in method `bar()`. The situation gets even worse when a medium priority thread T_m preempts thread T_l already executing within the synchronized block to execute its own method `foo()` (Figure 4.1). In general, the number of medium priority threads may be unbounded, making the time T_l remains preempted (and T_h blocked) unbounded as well, thus resulting in *unbounded priority inversion*. Such situations can cause havoc in applications where high-priority threads demand some level of guaranteed throughput.

Another problem related to using mutual exclusion, *deadlock*, has already been mentioned in one of the previous chapters. Deadlock results when two or more threads are

T_l T_h	T_m

`synchronized(mon) {`
`  o1.f++;`
`  o2.f++;`
`  bar();`
`}`
	`foo();`

Figure 4.1. Priority inversion

T	T'

`synchronized(mon1) {`
`  o1.f++;`
`  synchronized(mon2) {`
`  bar();`
`  }`
`}`
	`synchronized(mon2) {`
	`  o2.f++;`
	`  synchronized(mon1) {`
	`  foo();`
	`  }`
	`}`

Figure 4.2. Deadlock

unable to proceed because each is waiting to acquire a monitor held by another. Such a situation is easily constructed for two threads, T and T', as illustrated in Figure 4.2. Thread T acquires monitor `mon1` while T' acquires monitor `mon2`, then T tries to acquire `mon2` while T' tries to acquire `mon1`, resulting in deadlock. Deadlocks may also result from a far more complex interaction among multiple threads and may stay undetected until and beyond application deployment. The ability to resolve deadlocks dynamically is much more attractive than permanently stalling some subset of concurrent threads.

For real-world concurrent programs with complex module and dependency structures, it is difficult to perform an exhaustive exploration of the space of possible interleavings to determine statically when deadlocks or priority inversion may arise. When static techniques are infeasible, dynamic techniques can be used both to identify these problems and to remedy them whenever possible. Solutions to the unbounded priority inversion problem, such as the *priority ceiling* and *priority inversion* protocols [52] are examples of such dynamic solutions.

The priority ceiling technique raises the priority of any thread trying to acquire a monitor to the highest priority of any thread that ever uses that monitor (*i.e.*, its priority ceiling). This requires the programmer to supply the priority ceiling for each monitor used throughout the execution of a program. In contrast, priority inheritance will raise the priority of a thread only when holding a monitor causes it to block a higher priority thread. When this happens, the low priority thread inherits the priority of the higher priority thread it is block-

ing. Both of these solutions prevent a medium priority thread from blocking the execution of the low priority thread (and thus also the high priority thread) indefinitely. However, even in the absence of a medium priority thread, the high priority thread is forced to wait until the low priority thread releases its monitor. In the example presented in Figure 4.1, since the time to execute method `bar()` is potentially unbounded, high priority thread T_h may still be delayed indefinitely until low priority thread T_l finishes executing `bar()` and releases the monitor. Neither priority ceiling nor priority inheritance offer a solution to this problem. We are also not aware of any existing solutions that would enable dynamic resolution of deadlocks.

We use optimistic transactions as a foundation for a more general solution to resolving priority inversion and deadlock problems dynamically (and automatically, without changes to the language semantics) : *revocable monitors*. We retain the traditional model of managing concurrency control in Java, that is mutually exclusive monitors, and augment it with additional mechanisms originating in the realm of optimistic transactions.

4.1 Design

One of the main principles underlying the design of revocable monitors is complete *transparency*: programmers must perceive all programs executing in our system to behave exactly the same as on all other platforms implemented according to the Java Language Specification [23]. In order to achieve this goal we must adhere to Java's execution semantics [23, 38] and follow the Java Memory Model [43] access rules.

In both of the scenarios illustrated by Figures 4.1 and 4.2, one can identify one *offending* thread that is responsible for the occurrence of priority inversion or deadlock. For priority inversion the offending thread is the low-priority thread currently executing the monitor. For deadlock, it is either of the threads engaged in deadlock.

In a system using revocable monitors, every (outermost) synchronized block is executed as an optimistic transaction. When priority inversion or deadlock are detected, the transaction executed by the offending thread gets aborted and then subsequently re-started.

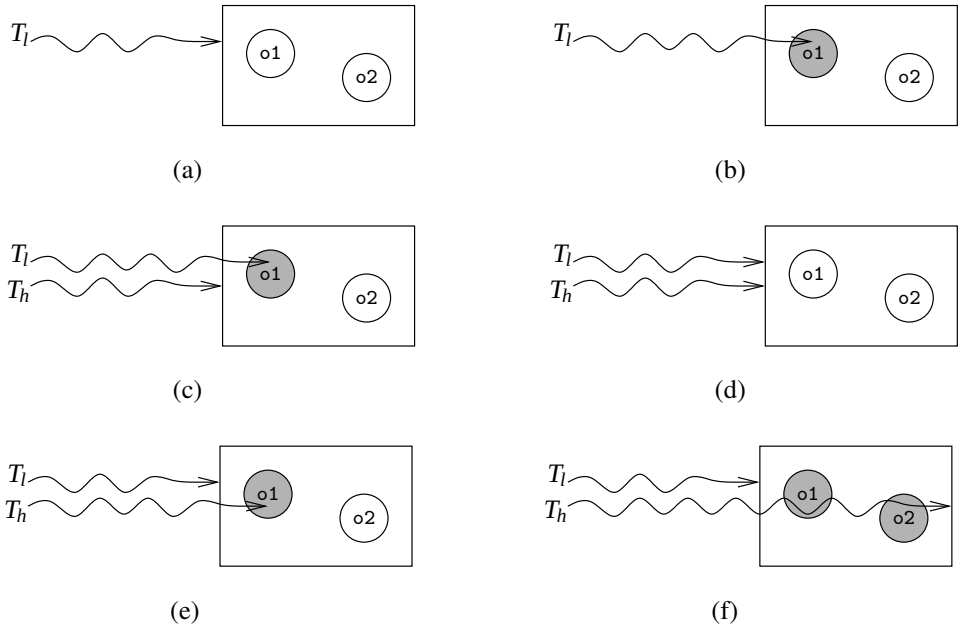


Figure 4.3. Resolving priority inversion

In other words, the monitor protecting the synchronized block and the transaction associated with the monitor get revoked – it appears as if the offending thread had never entered this section of code.

4.1.1 Resolving Priority Inversion and Deadlock

The design of revocable monitors deviates slightly from the traditional understanding of optimistic transactions, defined in terms of the three-phase approach, as described in Section 1.2.3. Because Java monitors are mutually exclusive, they already guarantee serializability during concurrent execution. Thus, instead of being re-directed to a log, updates can be performed "in-place" (as described in Section 2.2 discussing support for logging) and the validation phase can be omitted. Logging however is still required to support the process of undoing modifications performed within a region protected by the monitor being revoked.

The process of resolving priority inversion using revocable monitors is illustrated in Figure 4.3, where wavy lines represent threads T_l and T_h, circles represent objects `o1` and `o2`, updated objects are marked grey, and the box represents the dynamic scope of a common monitor guarding a synchronized block executed by the threads. This scenario is based on the code from Figure 4.1 (data access operations performed within method `bar()` have been omitted for brevity). In Figure 4.3(a), low-priority thread T_l is about to start a transaction and enter the synchronized block protected by monitor `mon`, which it does in Figure 4.3(b), modifying object `o1`. High-priority thread T_h tries to acquire the same monitor, but is blocked by T_l (Figure 4.3(c)). Here, a priority inheritance approach would raise the priority of thread T_l to that of T_h, but T_h would still have to wait for T_l to release the monitor. If a priority ceiling protocol was used, the priority of T_l would be raised to the ceiling upon its entry to the synchronized block, but the problem of T_h being forced to wait for T_l to release the monitor would remain. Instead, our approach revokes T_l's monitor `mon`: all updates to `o1` are undone, monitor `mon` is released, and T_l's synchronized block is re-executed. Thread T_l must then wait while T_h starts its own transaction, enters the synchronized block, updates objects `o1` (Figure 4.3(e)) and `o2` (Figure 4.3(f)), and commits the transaction after leaving the synchronized block. At this point the monitor is released and T_l will re-gain entry to the synchronized block. In the procedure described above, revocation of T_l's monitor logically re-schedules T_l to run after T_h.

The process of resolving deadlock is illustrated in Figure 4.4. The wavy lines represent threads T and T', circles represent objects `o1` and `o2`, updated objects are marked grey, and the boxes represent the dynamic scopes of monitors `mon1` and `mon2`. This scenario is based on the code from Figure 4.2. In Figure 4.4(a) thread T is about to enter the synchronized block protected by monitor `mon1`. In Figure 4.4(b) T acquires `mon1`, starts a transaction, updates object `o1` and attempts to acquire monitor `mon2`. In Figure 4.4(c) thread T_2 is about to enter the synchronized block protected by monitor `mon2`. In Figure 4.4(d) the same thread acquires `mon2`, starts a transaction, updates object `o2` and attempts to acquire monitor `mon1`. At this point both threads are deadlocked – both T and T' are blocked because each is waiting to acquire a monitor held by the other. In order

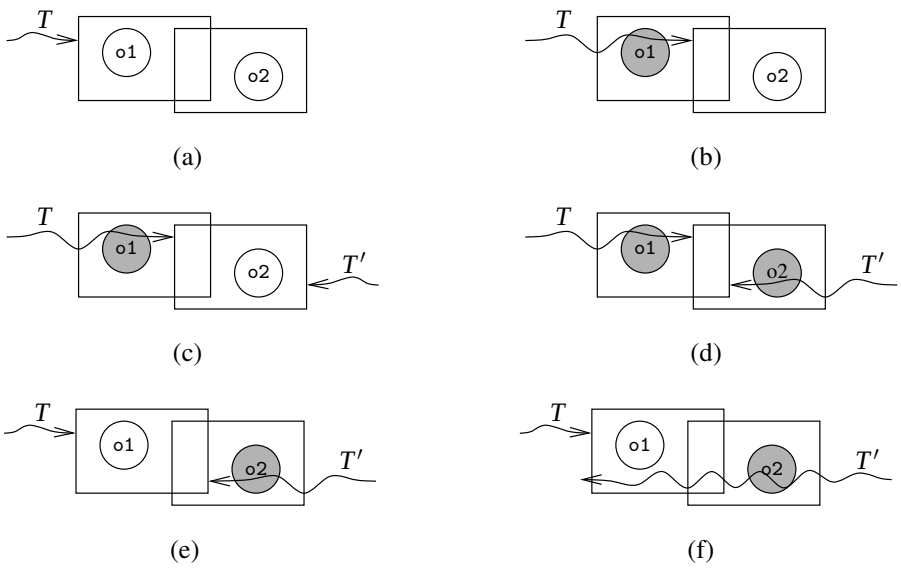


Figure 4.4. Resolving deadlock

to support deadlock detection, the run-time system may use a dynamically built *wait-for* graph [24] representing the waiting relationships between threads. Detection of any cycle in the wait-for graph (which can be done periodically by the run-time system) indicates existence of deadlock. Alternatively, time-outs may be used for the same purpose [24]. We assume that thread T's outermost monitor is selected for revocation: monitor `mon1` is released, all updates to `o1` are undone and execution of the synchronized block is re-tried. Thread T' may then acquire monitor `mon1`, proceed to execute method `foo()` (data access operations performed within method `foo()` have been omitted for brevity), release both monitor `mon1` and monitor `mon2` and commit its transaction (Figure 4.4(f)).

Some instances of deadlock cannot be resolved using revocable monitors. If deadlock is guaranteed to arise due to the way the synchronization protocol has been programmed (independently of scheduling) when using traditional non-revocable monitors, then the deadlock also cannot be resolved by revocable monitors. Consider the code fragment in Figure 4.5. Because of control-flow dependencies, all executions of this program under traditional mutual exclusion will eventually lead to deadlock. When executing this program using revocable monitors, the run-time system will attempt to resolve deadlock by

T	T'

`synchronized(mon1) {`
`  while (!o1.f) {`
`  synchronized(mon2) {`
`  bar();`
`  }`
`  }`
`  o2.f = true;`
`}`
	`synchronized(mon2) {`
	`  while (!o2.f) {`
	`  synchronized(mon1) {`
	`  foo();`
	`  }`
	`  }`
	`  o1.f = true;`
	`}`

Figure 4.5. Schedule-independent deadlock

revoking one of the threads' outermost monitors. Let's assume that thread T's outermost monitor is selected for revocation. In order for thread T' to make progress it must be able to observe updates performed by thread T which have not yet been executed. As a result, T' is unable to proceed – it will maintain ownership of the monitors it has already acquired, which will eventually lead to another deadlock once execution of thread T is resumed. Note however, that while revocable monitors are unable to assist in resolving schedule-independent deadlocks, the final observable effect of the resulting *livelock* (*i.e.*, repeated attempts to resolve the deadlock situation through aborts) is the same as for deadlock – none of the threads will make progress.

The introduction of revocations into the system requires a careful consideration of their interaction between with the Java Memory Model [43]. We elaborate on these issues in the following sections.

4.1.2 The Java Memory Model (JMM)

The JMM [43] defines a *happens-before* relation (written $\xrightarrow{hb}$) among the actions performed by threads in a given execution of a program. For single-threaded execution the happens-before relation is defined by program order. For multi-threaded execution a happens-before relation is induced between a release and a subsequent acquire opera-

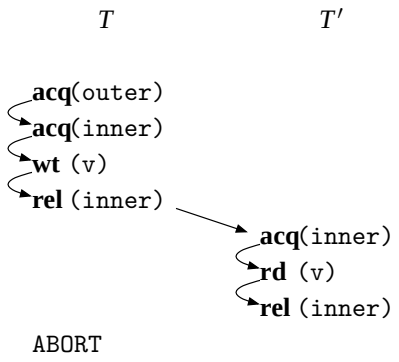


Figure 4.6. Revocation inconsistent with the JMM due to monitor nesting

tion on a given monitor mon. The happens-before relation is transitive: OP $\xrightarrow{hb}$ OP′ and OP′ $\xrightarrow{hb}$ OP″ imply OP $\xrightarrow{hb}$ OP″. The JMM shared data visibility rule is defined using the happens-before relation: a read **rd**(v) is *allowed* to observe a write **wt**(v) to a given variable v if **rd**(v) does not happen before **wt**(v) and there is no intervening write **wt′**(v) such that **rd**(v) $\xrightarrow{hb}$ **wt′**(v) $\xrightarrow{hb}$ **wt**(v) (we say that a read becomes *read-write dependent* on any write that it is *allowed* to see). In other words, for every pair of operations consisting of a read and a write, a programmer can rely on the read to observe the value of the write only if the read is read-write dependent. Considering these definitions we can conclude that it is possible for partial results computed by some thread T executing the synchronized block protected by monitor mon to become visible to (and to be used by) another thread T' even before thread T releases mon. This could happen if an operation executed by thread T before releasing mon induced a happens-before relation with an operation of thread T'. However, subsequent revocation of T's monitor would undo the update and remove the happens-before relation, making the value seen by T' appear "out of thin air" and thus make the execution of T' inconsistent with the JMM.

An example of such an execution appears in Figure 4.6 (arrows depict the happens-before relation), where execution of thread T's outermost monitor gets revoked at some point. Initially, thread T starts a transaction, acquires monitor outer and subsequently monitor inner, writes to a shared variable v and releases monitor inner. Then thread T' starts its own transaction, acquires monitor inner, reads variable v, commits the trans-

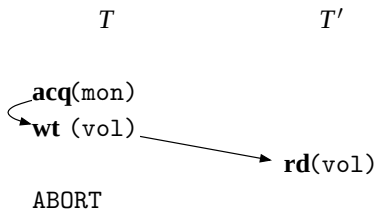


Figure 4.7. Revocation inconsistent with the JMM due to volatile variable access

action and releases monitor `inner`. The execution is JMM-consistent up to the point of abort: the read performed by T' is allowed but aborting the transaction executed by T is going to violate consistency with respect to the JMM.

A similar problem occurs when *volatile* variables are used. Volatile variables have different access semantics then "regular" variables. According to the JMM, there exists a happens-before relation between a volatile write and all subsequent volatile reads of the same (volatile) variable. For the execution presented in Figure 4.7 `vol` is a volatile variable and arrows depict the happens-before relation. As in the previous example, the execution is JMM-consistent up to the abort point because the read performed by T' is *allowed*, but the abort would violate consistency. We now discuss possible solutions to these consistency-preservation problems.

4.1.3 Preserving JMM-consistency

Several solutions to the problem of preserving JMM-consistency can be considered. We might trace read-write dependencies among all threads and upon revocation of a monitor trigger a cascade of revocations if read-write dependencies are violated. An obvious disadvantage of this approach is the need to consider *all* operations (including those performed outside of synchronized blocks) for potential revocation. In the execution of Figure 4.7 the volatile read performed by T' would have to be undone even though it is not guarded by any monitor. In general, the ability to undo an arbitrary part of a thread's computation could result in the execution of the entire thread to be transactional, which is

```
static boolean v=false;
```

T	T'

`synchronized(outer) {`
`  synchronized(inner) {`
`  v=true;`
`  }`
 	
`  // ABORT`
`}`
	`while (true) {`
	`  synchronized(inner) {`
	`  if (v) break;`
	`  }`
	`}`

Figure 4.8. Rescheduling thread execution in the presence of revocations may not always be correct

likely to incur considerable overhead. Furthermore, to apply this solution, the full execution context of each thread (*i.e.*, its instruction pointer, registers, thread stack *etc.*) would have to be available in the transaction's log in addition to its shared data operations. Consider a situation based on the same example (Figure 4.7) where thread T' *returns* (from the current method) after reading `vol` (and potentially using it to perform additional computation) but before thread T is asked to revoke its monitor `mon`. Without the ability to restore the full execution context of T', revocation of the effects of its own operation becomes infeasible.

Another possible solution is to re-schedule the execution of threads in problematic cases. In the examples of Figures 4.6 and 4.7, if thread T' executes fully before thread T, then the execution will still be JMM-consistent. Revocation of T's outermost monitor does not violate consistency since none of the updates performed by T are visible to T'. Besides the obvious question about the practicality of re-scheduling as a solution (some knowledge about the future actions performed by threads would be required), there also remains the issue of correctness. While re-scheduling may be correct in some cases, it is not necessarily correct in others. Consider the Java program of Figure 4.8. Completion of thread T' is dependent upon it seeing the effect of T executing the statement `v=true`. If we choose to reschedule T' to run before T, knowing that T's outermost monitor might be

revoked, then the execution of T' will never complete. Of course, if we make the "right" choice to reschedule T' after T, things will work. There are however, similar cases where rescheduling never works.

The solution that does seem flexible enough to handle all possible problematic cases, and simple enough to avoid using complex analyses and/or maintaining significant additional meta-data, is to disable the revocability of monitors whose revocation could create inconsistencies with respect to the JMM. As a consequence, not all instances of priority inversion or deadlock can be resolved. We mark a monitor, and thus a thread executing the synchronized block protected by this monitor, *non-revocable* when a read-write dependency is created between a write performed within the block[1] and a read performed by another thread. Detecting the possibility for this is relatively straightforward, without needing to track every read, so long as we track monitor acquire/release dependencies, as follows. When a thread holding an outer monitor acquires some inner monitor, it becomes *associated* with the inner monitor. This association is cleared when the thread releases the outer monitor, or when the thread is made non-revocable. Any other thread arriving at the monitor will simply make non-revocable any thread associated with that monitor, clearing the association. If the arriving thread itself holds an outer monitor then it now becomes associated with the monitor. A monitor is also marked non-revocable if it contains a write to a non-volatile variable. We believe this solution does not severely detract from the effectiveness of our technique. Intuitively, programmers protect accesses to the same subset of shared data using the same set of monitors; in such cases, there is no need to force non-revocability of any of the monitors (even if they are nested) since mutual exclusion induced by monitor acquisition prevents generation of problematic dependencies among these threads.

There exist other Java constructs that affect revocability of monitors. Calling a native method within a synchronized block also forces non-revocability of the monitor protecting the block (and all of the monitors protecting the enclosing synchronized blocks if it is nested), since the effects of a native method cannot generally be undone (*e.g.*, printing a

[1]The write may additionally be protected by other monitors nested within `mon`.

message to the console). The same applies to executions where a `wait` method is invoked within a nested monitor.[2] Revocation of the monitor protecting a `wait` call would result in a situation where the matching `notify` call (that "woke up" the waiting thread) "disappears" (*i.e.*, does not get delivered to any thread) which would violate Java execution semantics. A call to `notify` does not force irrevocability of enclosing monitors: Java permits "spurious wake-ups" [43] and a notification performed within a revoked monitor can be considered as such.

4.2 Implementation

When discussing the details of our implementation, we concentrate on describing compiler and run-time support necessary to support revocations. Because serializability is guaranteed by the presence of mutual exclusion, tracking of shared data dependencies is not required. For our case study we chose the priority inversion problem, rather than deadlock resolution, as an excellent vehicle to measure the trade-offs inherent in our approach.

4.2.1 Logging

Logging is realized through the use of read and write barriers, described in Section 2.4. Both compilers provided with the distribution of our implementation platform, Jikes RVM, have been modified to inject barriers for every store operation (represented by the following bytecodes: `putfield` for object stores, `Xastore` for array stores and `putstatic` for static variable stores. Once a thread starts a transaction and enters a synchronized block, the barriers record in the log every modification performed throughout execution of the entire transaction (until the transaction is either committed at the end of the synchronized block or aborted). The information stored in the log can then be used to undo operations on the shared data in case of the transaction's abort.

[2] A monitor object associated with the receiver object is released upon a call to `wait` and re-acquired after returning from the call. In the case of a non-nested monitor a potential revocation will therefore not reach beyond the point when `wait` was called.

We implemented the log as a sequential buffer. For object and array stores three values are recorded: the target object or array, the offset of the modified field or array slot, and the previous (old) value in that field/slot. For stores to static variable two values are recorded: the offset of the static variable in the global symbol table (*i.e.*, the JTOC in Jikes RVM) and the previous value of that variable.

4.2.2 Revocation

There exist two different synchronization constructs in Java: synchronized methods and synchronized blocks. We treat them uniformly, by transforming synchronized methods into non-synchronized equivalents whose entire body is enclosed in a synchronized block. For each synchronized method we create a non-synchronized wrapper with a signature identical to the original method. We fill the body of the wrapper method with a synchronized block enclosing an invocation of the original (non-synchronized) method, which has been appropriately renamed to avoid name clashes. We also instruct Jikes RVM to inline the original method within the wrapper to avoid performance penalties related to the delegation of method invocations. This approach greatly simplifies our implementation,[3] is extremely simple and robust, and also efficient because of inlining.

As described in Section 2.5, we use a modified version of the exception handling mechanism to support revocations. We use bytecode re-writing (supported by the Bytecode Engineering Library from Apache) to wrap each synchronized block with the appropriate exception handler and to inject code responsible for recording local state at the point when the synchronized block (and its respective transaction) starts. Since revocation may involve a nested synchronized block, each handler of the `Revoke` exception invokes an internal (virtual machine level) method to check if it corresponds to the synchronized block that needs to be re-executed. If it does, then the handler restores both shared and local state, releases the monitor protecting the synchronized block and returns control to

[3]We need only handle explicit *monitorenter* and *monitorexit* bytecodes, without worrying about implicit monitor operations for synchronized methods.

the beginning of this block. Otherwise, the handler re-throws the `Revoke` exception to the enclosing synchronized block, after releasing the monitor protecting the inner block.

Java supports the notion of default exception handlers. Such default handlers include both `finally` blocks, and `catch` blocks for exceptions of type `Throwable`, of which all exceptions (including `Revoke`) are instances. We must therefore modify the exception handling mechanism to prevent these handlers from being run while handling the `Revoke` exception, in order to preserve the transparency requirement (as described in Section 2.5). The default behavior still applies for all other exceptions, to preserve the standard semantics.

4.2.3 Priority Inversion Avoidance

Detecting priority inversion is reasonably simple. A thread acquiring a monitor deposits its priority in the header of the monitor object. Before another thread can acquire the monitor, the scheduler checks whether its priority is higher than the priority of the thread currently executing within the synchronized block. If it is, then the scheduler triggers revocation of the monitor held by the low priority thread, to allow its acquisition by the high-priority thread. The revocation is triggered by setting a flag checked by the low-priority thread at the subsequent yieldpoint[4] (the `Revoke` exception is thrown if the flag is set). If the incoming thread's priority is lower, it blocks on the monitor and waits for the other thread to complete execution of the synchronized block.

Jikes RVM does not include a priority scheduler; threads are scheduled in round-robin order. This does not affect the generality of our solution nor does it invalidate the results obtained, since the problems solved by our mechanisms cannot be solved simply by using a priority scheduler. However, in order to make the measurements independent of the random order in which threads arrive at a monitor, we augmented monitor queues to take priority into account. A thread can have either high or low priority. When a thread releases a monitor, another thread is scheduled from the queue. If it is a high-priority thread, it is

[4]Jikes RVM supports pseudo-preemptive thread scheduling. Thread switches are only possible at the explicit yieldpoints inserted by the compiler into the code-stream at pre-specified points (*e.g.*, loop back-edges).

allowed to acquire the monitor. If it is a low-priority thread, it is allowed to run only if there are no other high-priority threads waiting in the queue.

4.3 Experimental Evaluation

We quantify the overhead of the revocable monitors mechanism using a detailed micro-benchmark. We measure executions that exhibit priority inversion to verify if the increased overheads induced by our implementation are mitigated by higher overall throughput of high-priority threads. The experiments are performed for a uni-processor system. Since revocable monitors do nothing to increase concurrency in applications, applications will exhibit no more parallelism using revocable monitors on multi-processors than they would using non-revocable monitors. In our results, it is to be expected that revocable monitors used to address priority inversion will sacrifice throughput of low-priority threads to improve throughput of high-priority threads. As a result, total throughput will suffer. Our results quantify this sacrifice of total throughput to be approximately 30%, while throughput for high-priority threads improves by 20% to 160%.

4.3.1 Benchmark Program

The micro-benchmark executes several low-priority and high-priority threads contending on entry to the same synchronized block. Regardless of their priority, all threads are compiled identically, with write barriers inserted to log updates, and special exception handlers injected to support revocations of monitors. Though our benchmark is structured so that only monitors of low-priority threads will actually be revoked, updates of both low-priority and high-priority threads are logged for fairness, even though monitors of high-priority threads are never revoked. Every thread executes the synchronized block 100 times. The synchronized block contains an inner loop containing an interleaved sequence of read and write operations. We emphasize that our micro-benchmark has been constructed to gauge the overheads inherent in our techniques (the costs of re-execution, logging, *etc.*) and not necessarily to simulate any particular real-life application. We do

not bias the benchmark structure in favor of our mechanisms by artificially extending the execution time using benign (with respect to logging) operations (*e.g.*, method calls). We make the execution time of a synchronized block directly proportional to the number of shared data operations performed within that block. We fixed the number of iterations of the inner loop for low-priority threads at 500K, and varied it for the high-priority threads (100K and 500K). The remaining parameters for our benchmark include:

- The ratio of high-priority threads to low-priority threads – we used three configurations: $2+8$, $5+5$, and $8+2$, high-priority plus low-priority threads, respectively.

- The ratio of write to read operations performed within the synchronized block – we used six different configurations ranging from 0% writes (*i.e.*, 100% reads) to 100% writes (*i.e.*, 0% reads).

Our benchmark also includes a short random pause time (on average approximately a single thread quantum in Jikes RVM) right before entry to the synchronized block, to ensure random arrival of threads at the monitor protecting the block.

Our thesis is that the total elapsed time of high-priority threads can be improved using revocations, at the expense of longer elapsed time for low-priority threads. Improvement is measured against an implementation that provides no remedy for priority inversion. Thus, for every run of the micro-benchmark we compare the total time it takes for all high-priority threads to complete their execution for the following two settings:

- An *unmodified* version of Jikes RVM that does not allow revocation of monitors: when a high-priority thread wants to acquire the monitor already held by a low-priority thread it waits until the low-priority thread exits the synchronized block.

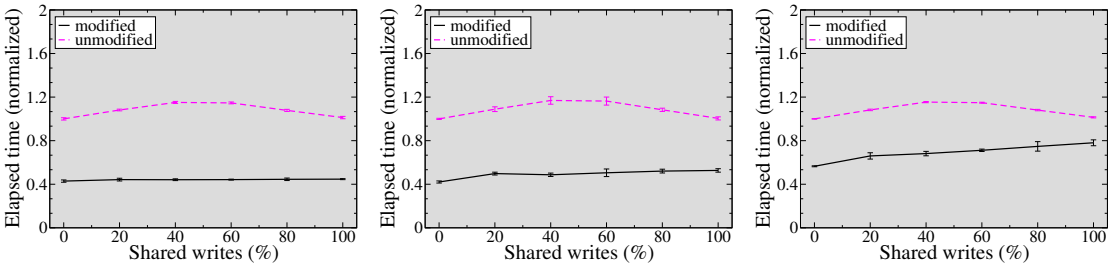
- A *modified* version of Jikes RVM equipped with the compiler and run-time changes to support revocation of monitors held by low-priority threads: when a high-priority thread wants to acquire a monitor held by a low-priority thread it signals its intent, resulting in the monitor held by the low-priority thread being revoked at the next yield point.

To measure the total elapsed time of high-priority threads we take two time-stamps for each high-priority thread: one when it begins its `run()` method and one at the end of its `run()` method. We compute the total elapsed time for all high-priority threads by subtracting the latest end time-stamp of all high-priority threads from the earliest begin time-stamp of all the high-priority threads. We also record the impact that our solution has on the overall elapsed time of the entire micro-benchmark, including low-priority elapsed times: this is simply the difference between the end time-stamp of the last thread to finish and the begin time-stamp of the first thread to start, regardless of priority.

The measurements were taken on an 800MHz Intel Pentium III (Coppermine) with 1GB of RAM running Linux kernel version 2.4.20-13.7 (RedHat 7.0). A benchmark run consists of one invocation of the VM in which the benchmark is repeated six times. We discard the results of the first iteration, in which the benchmark classes are loaded and compiled, to eliminate the overheads of compilation. We report the average elapsed time for the five subsequent iterations, and show 90% confidence intervals in our results. Our system is based on Jikes RVM 2.2.1 and we use a configuration where both the VM (which is itself implemented and bootstrapped in Java) and dynamically loaded classes are compiled using the optimizing compiler by default. Even in this configuration there remain some methods (*e.g.*, class initializers) that override this setting and are compiled without optimization.

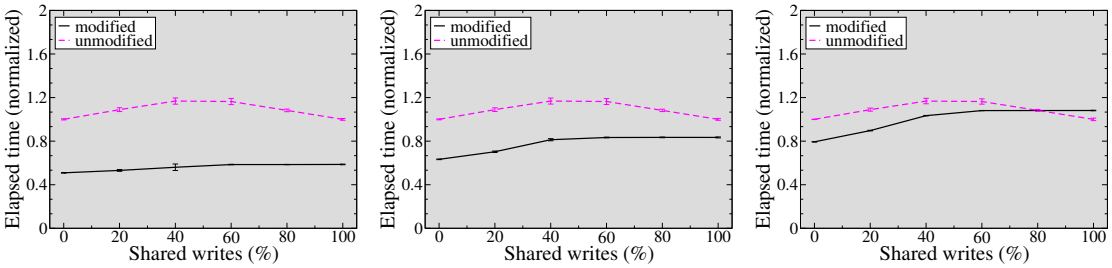
4.3.2 Results

Figures 4.9 and 4.10 plot elapsed times for high priority threads executed on both the modified version of Jikes RVM (indicated by a solid line) and the unmodified one (indicated by a dotted line), normalized with respect to the configuration executing 100% reads on the unmodified version (using standard non-revocable monitors). We normalize with respect to the 100% reads benchmark configuration so as to obtain a standard baseline for illustrating performance trends as the read/write mix changes. In Figure 4.9 every high priority thread executes 100K internal iterations; in Figure 4.10 the iteration count



(a) 2 high-priority, 8 low-priority (b) 5 high-priority, 5 low-priority (c) 8 high-priority, 2 low-priority

Figure 4.9. Total time for high-priority threads, 100K iterations



(a) 2 high-priority, 8 low-priority (b) 5 high-priority, 5 low-priority (c) 8 high-priority, 2 low-priority

Figure 4.10. Total time for high-priority threads, 500K iterations

is 500K. In each figure: the graph labeled (a) reflects a workload consisting of two high-priority threads, and eight low-priority threads; the graph labeled (b) reflects a workload consisting of five high-priority and five low-priority threads; and, the graph labeled (c) reflects a workload consisting of eight high-priority threads and two low-priority ones.

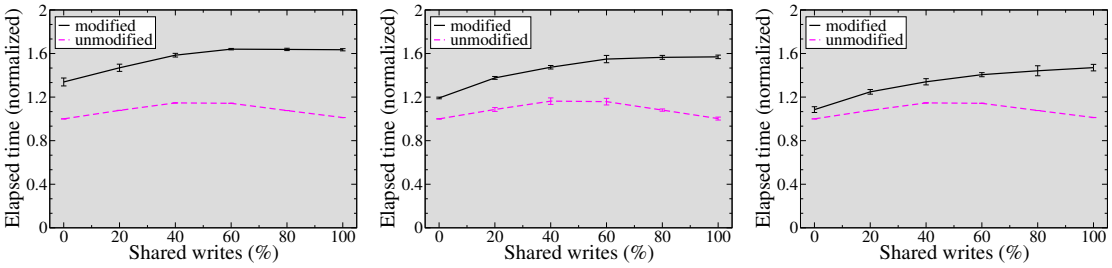
If the ratio of high-priority threads to low-priority threads is relatively low (Figures 4.9-4.10 (a)(b)), the modified version of Jikes RVM improves throughput for high-priority threads by 20% to 160% over the unmodified one. Average elapsed-time gain across all the configurations, including those where the number of high-priority threads is greater than the number of low-priority threads, is 78%. If we discard the configuration where there are eight high-priority threads competing with only two low-priority ones, the average elapsed time of a high-priority thread is half that of the execution time for the reference (unmodified) version.

Note that the influence of different read-write ratios on overall performance is small; recall that all threads, regardless of their priority, log all updates within a synchronized block. This implies that the cost of operations related to log maintenance and undoing of partial results is also small, compared to the elapsed time of the entire benchmark. Indeed, the actual "workload" (the contents of the synchronized block) in the benchmark consists entirely of data access operations – no delays (method calls, empty loops, *etc.*) are inserted in order to artificially extend its execution time. Since realistic programs are likely to have a more diverse mix of operations, the overheads would be even smaller in practice.

As expected, if the number of write operations within the synchronized block is sufficiently large, the overhead of logging and roll-backs may start outweighing potential benefit. For example, in Figure 4.10(c), under a 100% write configuration, every high priority thread writes, and thus logs, approximately 500K words of data in every execution of a synchronized block. We believe that synchronized blocks that consist entirely of write operations of this magnitude are relatively rare.

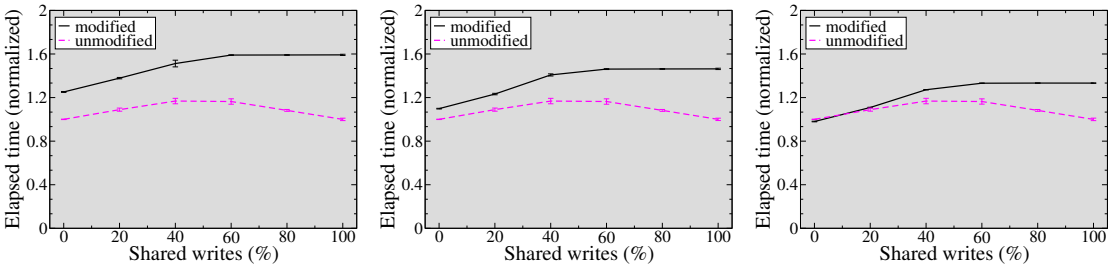
As the ratio of high-priority threads to low-priority threads increases, the benefit of our strategy diminishes (see Figures 4.9(c) and 4.10(c)). This is expected; since there are relatively fewer low-priority threads in the system, there is less opportunity to "steal" cycles from them to improve throughput of higher priority ones. We note, however, that even when the version of Jikes RVM using revocations has weaker performance than the unmodified implementation, the average difference in execution time is only a few percent.

Figures 4.11 and 4.12 plot overall elapsed times for the entire application executed on both modified (solid line) and unmodified (dotted line) versions of Jikes RVM. These graphs are also normalized with respect to a configuration executing 100% reads on the unmodified VM. Note that the overall elapsed time for the modified version of Jikes RVM must always be longer than for the unmodified one. If we disallowed revocability of monitors, threads executing on both versions of the VM would need exactly the same amount of time to execute their workloads (modulo costs related to the implementation of our mechanisms for the modified VM such as barriers, log maintenance, *etc.*). However, if the execution of monitors can be revoked, low-priority threads executing on the modified



(a) 2 high-priority, 8 low-priority (b) 5 high-priority, 5 low-priority (c) 8 high-priority, 2 low-priority

Figure 4.11. Overall time, 100K iterations



(a) 2 high-priority, 8 low-priority (b) 5 high-priority, 5 low-priority (c) 8 high-priority, 2 low-priority

Figure 4.12. Overall time, 500K iterations

VM will re-execute parts of their synchronized blocks, thus lengthening overall elapsed time. Since our focus is on lowering elapsed times for high priority threads, we consider the impact on overall elapsed time (on average 30% higher on the modified VM) to be acceptable. If our mechanism is used to resolve deadlocks then these overheads may be an even more acceptable price to pay to obtain progress by breaking deadlocks.

4.4 Related Work

A solution close in spirit to revocable monitors has been subsequently proposed by Manson *et al.* [42]. Their *preemptible atomic regions* (*PAR*s) are an extension to the Real-Time Specification for Java [11]. PARs address the problems of priority inversion and deadlock in real-time concurrent programming, focusing on providing real-time guarantees, such as the ability to compute worst-case execution time for all methods executing

in the system. Their implementation has been realized in the Ovm real-time Java virtual machine [2] for a uni-processor setting. Similarly to our own solution, PARs are executed as lightweight transactions. However, only one transaction can be present in the system at any given time. A thread that is executing a PAR gets immediately aborted once another thread attempts to start executing its own PAR. This guarantees the desired atomicity property – there is no interference between operations of threads executing different PARs and no thread executing a PAR can observe partial effects produced by another thread executing a different PAR. Updates performed by threads within PARs operate directly on shared data – they can be undone using information recorded in the sequential log if a transaction needs to be aborted. Aborted transactions are automatically re-executed (support for re-execution is provided through bytecode re-writing). Because only one PAR can execute at any given time, only a single instance of the log needs to be present in the system. In order to satisfy the real-time requirements, the size of the log must be bounded. Programmers are responsible for setting the size of the log a priori, in order to allow computation of the worst-case execution time for methods containing PARs. PARs are backwards compatible with the original formulation of the RTSJ – programs can use a mixture of PAR-based and traditional synchronization. The experimental results demonstrate improvements in both predictability of the response time and the overall throughput of the high-priority threads, when compared to solutions based on mutual exclusion.

Earlier in this chapter, we have discussed other, more traditional, dynamic techniques to handle priority inversion, such as the priority inheritance and the priority ceiling protocols. However, apart from the solution presented by Manson *et al.* and more general transactional schemes presented in Chapter 3, very few dynamic solutions to handle deadlock exist.

Zeng [16] proposes to treat deadlock occurrences as run-time exceptions. In his system, a broader than usual definition of deadlock is assumed, including situations when a thread is blocked while waiting for a notification. Therefore, in addition to using a wait-for graph (describing waiting relationships between threads), time-outs are used to detect deadlock in his system. Detection of deadlock is signaled by throwing a run-time excep-

tion which encodes information that can subsequently be used by an application-specific (provided by the programmer) deadlock resolution procedure. The proposed scheme has been implemented in the Latte Java Virtual Machine [1] but no empirical evidence concerning the related overheads has been provided.

Zeng and Martin [17] describe a technique to prevent occurrences of deadlocks in Java dynamically. At run-time they construct a lock-order graph which records the order in which mutual-exclusion locks are acquired globally by the application. Cycles in the graph represent sets of locks that can potentially form a deadlock. In order to prevent deadlock from happening, any thread trying to acquire a lock from the cyclic lock-set must first (implicitly) acquire a *phantom lock* – a special lock designated by the run-time system and associated with every cyclic lock-set. Acquisition of the phantom lock guarantees that only one thread can hold any lock from the cyclic lock-set at any given time and thus no deadlock involving any of these locks can occur. Their implementation of the deadlock prevention scheme in the Latte Java Virtual Machine exhibits low run-time overheads (on the order of 3%), but cannot prevent deadlocks that occur during construction of the lock-order graph. The effects that the introduction of phantom locks may have on the JMM visibility rules have not been discussed in their work.

4.5 Conclusions

In this chapter we have described how optimistic transactions can be adopted to solve priority inversion and deadlock problems. We have demonstrated the effectiveness of our approach in improving throughput of high-priority threads in a priority scheduling environment. Our experiments indicate that throughput for high-priority threads can be improved by 20% to 160% at a cost of approximately 30% total throughput loss.

5 SAFE FUTURES

A *future* is a simple and elegant concurrency abstraction, introduced for the first time in MultiLisp [25]. The MultiLisp `future` keyword is used to annotate expressions to have them evaluated concurrently (asynchronously) with the rest of the program. Such an annotated expression (*i.e.*, future) returns a placeholder that ultimately holds the value yielded by the expression. The result of the future's evaluation can be retrieved at a later time from the placeholder by *claiming* the future. This operation serves to synchronize the asynchronous evaluation of the future with the part of the program performing the claim.

Futures are an elegant alternative to programming with explicit threads because they often allow concurrent programs to be created through a relatively small rewrite of its sequential counterpart. Furthermore, in the absence of side-effects, futures satisfy a simple *safety* property: if a sequential program P is annotated with futures to yield concurrent program P_F, then the observable behavior of P is equivalent to P_F. Indeed, because futures are provided as expression annotations, their effect is intended to be fully transparent, visible only in the form of improved concurrency, without altering the meaning of the original sequential program.

Recently, futures have been introduced to Java in the form of the new interface specified by the `java.util.concurrent` package [34] as part of the Java 2 Platform Standard Edition 5.0 API. In Java, futures become specially designated method calls that can be evaluated concurrently with the rest of the program. However, in the presence of mutation, the safety property for futures no longer holds. A task spawned to evaluate a future may perform updates of shared data concurrently accessed by other tasks, including the task that spawned it. While this is not a serious issue in functional or mostly-functional languages where updates to shared data occur infrequently (if at all), it is significantly more problematic in Java where computation is typically structured in terms of modification to shared objects.

Consider a future f that executes concurrently with the task C_f that evaluates f's *continuation*. A continuation of a future is the computation that logically follows it. Safe execution of f may be compromised if it observes the effects of operations performed by C_f; for example, if C_f updates an object that is subsequently read by f. Similarly, safe execution of C_f may be compromised if it accesses an object that is subsequently written by f. Both these cases lead to different behavior than if the future and its continuation were evaluated sequentially. We believe that many of the notable benefits from using Java futures are significantly weakened by this lack of transparency with respect to access to shared data. Currently, in order to achieve some measure of safety, programs using futures must be further refined to provide explicit synchronization on potentially shared objects.

Our solution, *safe futures*, automatically preserves all the desired safety invariants. We define semantics to formalize our notion of safety by imposing constraints on the set of schedules that can be generated by a program in which concurrency is expressed exclusively through the use of futures. We design and and implement safe futures using the same mechanisms as those underlying optimistic transactions. We track shared data dependencies to detect safety violations and revoke executions upon detection of such violations. We now proceed to define semantics for safe futures and to formally argue correctness of our solution.

5.1 Semantics

To examine notions of safety with respect to interleavings of actions that operate within a future and its continuation, we define semantics for a call-by-value object calculus similar to Classic Java [22] extended with threads and a `future` construct. The semantics yield a *schedule* – a sequence of read and write operations performed during the execution of a program. A schedule is *serial* when all the operations of a program are executed within a single (main) thread. A schedule is *concurrent* if fragments of a program are executed concurrently by separate threads; in this case, the actions of these threads may be interleaved with one another. We impose safety conditions on concurrent schedules to

$$
\begin{array}{rcl}
P & ::= & (P \mid P) \mid \mathtt{t}[\mathtt{e}]_{\mathtt{l}} \\
L & ::= & \mathtt{class}\ \mathtt{C}\{\overline{\mathtt{f}}\ \overline{\mathtt{M}}\} \\
\mathtt{M} & ::= & \mathtt{m}\,(\overline{\mathtt{x}})\{\,\mathtt{e}\,\} \\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{l} \mid \mathtt{this} \mid \mathtt{e.f} \mid \mathtt{e.f} := \mathtt{e} \mid \mathtt{e.m}\,(\overline{\mathtt{e}}) \\
 & \mid & \langle \mathtt{e} \rangle \mathtt{e} \mid \mathtt{new}\ \mathtt{C}\,() \mid \mathtt{future}\,(\mathtt{e}) \mid \mathtt{get}\,(\mathtt{e})
\end{array}
$$

Figure 5.1. Language syntax.

verify that operation interleavings do not violate safety invariants. Informally, a concurrent schedule is safe if it is equivalent, in terms of its actions on shared data, to some serial schedule.

The syntax of the calculus is presented in Figure 5.1, its semantics in Figures 5.2– 5.3. A program defines a collection of class definitions, and a collection of threads. Classes are all uniquely named, and define a collection of instance fields and instance methods that operate over these fields. Every method consists of an expression whose value is returned as the result of a call to that method. An expression is either a variable, a location that references an object, the pseudo-variable `this`, a field reference, an assignment, a method invocation, a sequencing operation, an object creation operation, a `future` creation, or a `get` expression that claims a future.

Every class has a unique (nullary) constructor to initialize object fields. The application of a constructor returns a reference to an object instantiated from the class definition. A value is either `null`, an object instantiated from a class containing locations for the fields declared by the class, or a location that serves as a placeholder to hold the result of evaluating a future. A thread is uniquely labeled with a thread identifier, and a placeholder location.

We take metavariables L to range over class declarations, `C` to range over class names, `M` to range over methods, `m` to range over method names, `f` and `x` to range over fields and parameters, respectively, `l` to range over locations, and `v` to range over object values. We also use P for process terms, and `e` for expressions. We use over-bar to represent a finite ordered sequence, for instance, $\overline{\mathtt{f}}$ represents $\mathtt{f}_1\,\mathtt{f}_2\,\ldots\,\mathtt{f}_n$. The term $\overline{\alpha}\alpha$ denotes the exten-

Evaluation contexts: **Program states:**

$$
\begin{array}{rcl}
\mathtt{E} & ::= & \bullet \\
& | & \mathtt{E}[\bullet].\mathtt{f} := \mathtt{e} \\
& | & \mathtt{l}.\mathtt{f} := \mathtt{E}[\bullet] \\
& | & \mathtt{E}[\bullet].\mathtt{m}(\overline{\mathtt{e}}) \\
& | & \mathtt{l}.\mathtt{m}(\overline{\mathtt{l}}\ \mathtt{E}[\bullet]\ \overline{\mathtt{e}}) \\
& | & \mathtt{E}[\bullet]\ ;\ \mathtt{e} \\
& | & \mathtt{get}(\mathtt{E}[\bullet])
\end{array}
$$

$$\mathtt{E}^{\mathtt{t}}_{P}[\mathtt{e}]_{\mathtt{l}} ::= P \mid \mathtt{t}[\mathtt{E}[\mathtt{e}]]_{\mathtt{l}}$$

$$
\begin{array}{rcll}
\mathtt{t} & \in & \textit{Tid} & \\
P & \in & \textit{Process} & \\
\mathtt{x} & \in & \textit{Var} & \\
\mathtt{l} & \in & \textit{Loc} & \\
\mathtt{v} & \in & \textit{Val} & = \ \mathtt{null} \mid \mathtt{C}(\overline{\mathtt{l}}) \mid \mathtt{l} \\
\Gamma & \in & \textit{Store} & = \ \textit{Loc} \rightarrow \textit{Val} \\
\mathrm{OP}_{\mathtt{t}}\,\mathtt{l} & \in & \textit{Ops} & = \ \{\mathbf{rd}, \mathbf{wt}\} \times \textit{Tid} \times \textit{Loc} \\
S & \in & \textit{Schedule} & = \ \overline{\mathrm{OP}_{\mathtt{t}}\,\mathtt{l}} \\
\Lambda & \in & \textit{State} & = \ \textit{Process} \times \textit{Store} \times \textit{Schedule}
\end{array}
$$

Figure 5.2. Program states and evaluation contexts.

sion of the sequence $\overline{\alpha}$ with a single element α, and $\overline{\alpha}\overline{\alpha}'$ denotes sequence concatenation, $S.\mathrm{OP}_{\mathtt{t}}$ denotes the extension of schedule S with operation $\mathrm{OP}_{\mathtt{t}}$.

Program evaluation and schedule construction are specified by a global reduction relation, $P,\Gamma,S \Longrightarrow P',\Gamma',S'$, that maps a program state to a new program state. A program state consists of a collection of evaluating threads (P), a global store (Γ) to map locations to values, and schedules (S) to define a global interleaved sequence of actions performed by threads. Local reductions within a thread are specified by an auxiliary relation, $\mathtt{e},\Gamma,S \rightarrow_{\mathtt{t}} \mathtt{e}',\Gamma',S'$ that evaluates expression $\mathtt{e}$ within thread $\mathtt{t}$ to a new expression $\mathtt{e}'$; in doing so, a new store, and schedule may result. The only actions that are recorded by a schedule are those that read and write locations. The interpretation of schedules with respect to safety is the topic of the next section.

We use evaluation contexts to specify order of evaluation within a thread, and to prevent premature evaluation of the expression encapsulated within a `future` annotation. We define a process context $\mathtt{E}^{\mathtt{t}}_{P}[\mathtt{e}]_{\mathtt{l}}$ to denote an expression $\mathtt{e}$ available for execution by thread $\mathtt{t} \in P$ in a program state; the label $\mathtt{l}$ denotes a placeholder location that holds the result of $\mathtt{e}$'s evaluation.

The sequential evaluation rules are standard: holes in evaluation contexts can be replaced by the value of the expression substituted for the hole, sequence operations evalu-

Sequential evaluation rules:

$$\frac{\begin{array}{c}\mathtt{l}',\overline{\mathtt{l}}\ \textit{fresh}\\ \Gamma' = \Gamma[\mathtt{l}' \mapsto \mathtt{C}(\overline{\mathtt{l}}), \overline{\mathtt{l}} \mapsto \mathtt{null}]\\ S' = S.\mathbf{wt}_{\mathtt{t}}\,\mathtt{l}_1.\ldots.\mathbf{wt}_{\mathtt{t}}\,\mathtt{l}_n.\mathbf{wt}_{\mathtt{t}}\,\mathtt{l}'\\ \mathtt{l}_1,\ldots,\mathtt{l}_n \in \overline{\mathtt{l}}\end{array}}{\mathtt{new\ C}\,(),\Gamma,S \rightarrow_{\mathtt{t}} \mathtt{l}',\Gamma',S'} \qquad \frac{\begin{array}{c}\Gamma(\mathtt{l}) = \mathtt{C}(\overline{\mathtt{l}''}) \qquad \Gamma(\mathtt{l}') = \mathtt{v}\\ \Gamma' = \Gamma[\mathtt{l}''_i \mapsto \mathtt{v}]\\ S' = S.\mathbf{rd}_{\mathtt{t}}\,\mathtt{l}'.\mathbf{wt}_{\mathtt{t}}\,\mathtt{l}''_i\end{array}}{\mathtt{l.f}_i := \mathtt{l}',\Gamma,S \rightarrow_{\mathtt{t}} \mathtt{l}',\Gamma',S'}$$

$$\frac{}{\langle \mathtt{l} \rangle \mathtt{e},\Gamma,S \rightarrow_{\mathtt{t}} \mathtt{e},\Gamma,S} \qquad \frac{\begin{array}{c}\mathtt{class\ C}\,\{\overline{\mathtt{f}}\ \overline{\mathtt{M}}\} \in L \qquad \Gamma(\mathtt{l}) = \mathtt{C}(\overline{\mathtt{l}'})\\ S' = S.\mathbf{rd}_{\mathtt{t}}\,\mathtt{l}'_i\end{array}}{\mathtt{l.f}_i,\Gamma,S \rightarrow_{\mathtt{t}} \mathtt{l}'_i,\Gamma,S'}$$

$$\frac{\begin{array}{c}\Gamma(\mathtt{l}) = \mathtt{C}(\overline{\mathtt{l}'}) = \mathtt{v} \qquad \Gamma(\overline{\mathtt{l}}) = \overline{\mathtt{v}'}\\ \mathtt{class\ C}\,\{\overline{\mathtt{f}}\ \overline{\mathtt{M}}\} \in L \qquad \mathtt{m}\,(\overline{\mathtt{x}})\,\{\,\mathtt{e}\,\} \in \overline{\mathtt{M}}\end{array}}{\mathtt{l.m}\,(\overline{\mathtt{l}}),\Gamma,S \rightarrow_{\mathtt{t}} [{}^{\mathtt{v}}/_{\mathtt{this}}, {}^{\overline{\mathtt{v}'}}/_{\overline{\mathtt{x}}}]\mathtt{e},\Gamma,S}$$

Global evaluation rules:

$$\frac{\mathtt{e},\Gamma,S \rightarrow_{\mathtt{t}} \mathtt{e}',\Gamma',S'}{\mathtt{E}^{\mathtt{t}}_P[\mathtt{e}]_{\mathtt{l}},\Gamma,S \Longrightarrow \mathtt{E}^{\mathtt{t}}_P[\mathtt{e}']_{\mathtt{l}},\Gamma',S'} \qquad \frac{P = P' \mid \mathtt{t}'[\mathtt{l}'']_{\mathtt{l}'}}{\mathtt{E}^{\mathtt{t}}_P[\mathtt{get}\ (\mathtt{l}')]_{\mathtt{l}},\Gamma,S \Longrightarrow \mathtt{E}^{\mathtt{t}}_P[\mathtt{l}'']_{\mathtt{l}},\Gamma,S}$$

$$\frac{\mathtt{t}',\mathtt{t}''\ \textit{fresh} \qquad \mathtt{t} \leq \mathtt{t}' \leq \mathtt{t}'' \qquad \mathtt{l}'\ \textit{fresh}}{\mathtt{E}^{\mathtt{t}}_P[\mathtt{future}\ (\mathtt{e})]_{\mathtt{l}},\Gamma,S \Longrightarrow P \mid \mathtt{t}'[\mathtt{e}]_{\mathtt{l}'} \mid \mathtt{t}''[\mathtt{E}[\mathtt{l}']]_{\mathtt{l}},\Gamma,S}$$

Figure 5.3. Language semantics.

ate left-to-right, method invocation evaluates the method body in the original environment augmented by binding actuals to parameters in addition to binding the pseudo-variable `this` to the current receiver object. Read and write operations augment the schedule in the obvious way. A `new` expression extends the schedule with writes to all instance fields (with null values).

An expression of the form `future (e)` causes `e`'s evaluation to take place in a new thread $\mathtt{t}'$. A fresh location $\mathtt{l}'$ is created as a placeholder to hold the result of evaluating

this future. Thus, $\mathtt{t}'[\mathtt{e}]_{\mathtt{l}'}$ denotes a thread with identifier $\mathtt{t}'$ that evaluates expression e and stores the result of this evaluation into $\mathtt{l}'$.

In addition to the thread responsible for computing the value of the future, a new thread $\mathtt{t}''$ is created to evaluate the future's continuation. As a result, the parent thread is no longer relevant. This specification simplifies the safety conditions discussed below. The thread identifiers associated with threads created by a future expression are related under a total ordering ($\leq$). Informally, this ordering captures the logical (sequential) order in which actions performed by the threads must be evaluated. Thus, if $\mathtt{t}' \leq \mathtt{t}''$, then either $\mathtt{t}' = \mathtt{t}''$, or all actions performed by $\mathtt{t}'$ must logically take place before $\mathtt{t}''$. In particular, effects induced by actions performed by $\mathtt{t}''$ must not be visible to operations in $\mathtt{t}'$.

Synchronization takes place through the get expression, being the equivalent of the claim operation in the original formulation of futures [25]. In the rule for get, the location label $\mathtt{l}'$ represents a placeholder or synchronization point that holds the value of a task spawned by a future. The rule is satisfied precisely when the associated future (say, future (e)) has completed. When this occurs, the process state will contain a thread with shape $\mathtt{t}[\mathtt{l}'']_{\mathtt{l}'}$ where $\mathtt{l}''$ is the location yielded by evaluation of e.

5.1.1 Safety

A schedule defines a sequence of possibly interleaved operations among threads. The correctness of a schedule, therefore, must impose safety constraints on read and write operations. These constraints guarantee that the injection of futures into an otherwise sequential program does not alter the meaning of the program. Thus, these constraints must ensure that interleavings are benign with respect to read and write operations. The semantics does not permit reordering of operations within a thread.

There are two conditions (roughly equivalent to the Bernstein conditions [7]) that must hold on schedules to guarantee this property: (1) an access to a location *l* (either a read or a write) performed by a future should not witness a write to l performed earlier by its continuation, and (2) a write operation to some location l performed by a future should be

visible to the first access (either a read or a write) made to `l` by its continuation. In other words, no write to a location `l` by a future's continuation can occur before any operations on `l` by the future, and all writes to a location `l` by the future must occur before any operation to `l` by the continuation. Note that these conditions do not prohibit interleaved operations by a future and its continuation to distinct locations.

We summarize these constraints in terms of two safety rules, *csafe* and *fsafe*, resp. The former captures the notion of when an operation performed by a continuation is safe with respect to the actions performed by the future within a schedule, and the latter captures the notion of when an operation performed by a future is safe with respect to its continuation within a schedule.

$$\frac{\mathbf{wt}_{\mathtt{t}'}\,\mathtt{l}, \mathbf{rd}_{\mathtt{t}'}\,\mathtt{l} \notin S', \quad \mathtt{t}' \leq \mathtt{t}}{\mathit{csafe}(S.\mathbf{wt}_{\mathtt{t}}\,\mathtt{l}.S')} \qquad \frac{\mathbf{wt}_{\mathtt{t}'}\,\mathtt{l}, \mathbf{rd}_{\mathtt{t}'}\,\mathtt{l} \notin S, \quad \mathtt{t} \leq \mathtt{t}'}{\mathit{fsafe}(S.\mathbf{wt}_{\mathtt{t}}\,\mathtt{l}.S')}$$

Definition 5.1.1 *(Schedule Safety)*
A schedule S is safe if csafe(S) *and fsafe*(S) *hold.*

To validate the safety of an interleaved schedule, we must ensure that its observable behavior is equivalent to the behavior of a corresponding program in which futures have no computational effect. In such a program, evaluation of the future's continuation is delayed until the future itself is fully evaluated. This trivially enforces sequential order between all operations executed by the future and all operations executed by the continuation and thus automatically yields a serial schedule.

We first introduce the notion of a schedule *permutation* that allows us to define an equivalence relation on schedules:

Definition 5.1.2 *(Permute) Schedule S is a* permutation *of schedule S′ (written* $S \leftrightarrow S'$*), if* $\mathit{len}(S) = \mathit{len}(S')$ *and for every* $\mathrm{OP}_{\mathtt{t}}\,\mathtt{l}_i \in S$*, there exists a unique* $\mathrm{OP}_{\mathtt{t}}\,\mathtt{l}_j \in S'$*.*

A *serial* schedule is a schedule in which no interleaving among operations of different threads occurs:

Definition 5.1.3 *(Serial Schedule)*

Schedule $S = \text{OP}_{\mathtt{t}_1}\,\mathtt{l}_1.\ldots.\text{OP}_{\mathtt{t}_n}\,\mathtt{l}_n$ *is serial if for all* $\text{OP}_{\mathtt{t}_j}\,\mathtt{l}_j$ *there does not exist* $\text{OP}_{\mathtt{t}_k}\,\mathtt{l}_j$, $k > j$ *such that* $\mathtt{t}_k < \mathtt{t}_j$.

We wish to show that any safe schedule can be permuted to a serial one since a serial schedule reflects an execution in which operations executed by a future are not interleaved with operations performed by its continuation. Effectively, a serial schedule reflects an execution in which a spawned future runs to completion before any operations in its continuation are allowed to execute; in other words, a serial schedule corresponds to a program execution in which futures have no computational effect.

We first appeal to a lemma that allows us to permute adjacent operations belonging to different threads in a safe schedule:

Lemma 5.1.1 *(Permutation)*

Let schedule $S = \text{OP}_{\mathtt{t}_1}\,\mathtt{l}_1.\text{OP}_{\mathtt{t}_2}\,\mathtt{l}_2$ *be safe. Then if S is safe, there exists a serial schedule* S' *such that* $S \leftrightarrow S'$.

Proof If $\mathtt{t}_1 \leq \mathtt{t}_2$, then the schedule is trivially serial. If $\mathtt{t}_1 > \mathtt{t}_2$, and because S is safe, it must be the case that either (a) $\mathtt{l}_1 \neq \mathtt{l}_2$, or (b) $\mathtt{l}_1 = \mathtt{l}_2 = \mathtt{l}$, and $\text{OP}_{\mathtt{t}_2}\,\mathtt{l} = \mathbf{rd}_{\mathtt{t}_2}\,\mathtt{l}$. In both cases, we can choose $S' = \text{OP}_{\mathtt{t}_2}\,\mathtt{l}_2.\text{OP}_{\mathtt{t}_1}\,\mathtt{l}_1$.

Our soundness result generalizes this lemma over schedules of arbitrary length:

Theorem 5.1.1 *(Soundness)*

If schedule S is safe, then there exists a serial schedule S' *such that* $S \leftrightarrow S'$.

Proof The proof is by induction on schedule length. Lemma 1 satisfies the base case. Suppose $S = S_1.\text{OP}_{\mathtt{t}}\,\mathtt{l}$ where $\mathit{len}(S_1) > 2$. By the induction hypothesis, there exists a serial schedule S_1' such that $S_1' \leftrightarrow S_1$. Suppose $S_1' = \text{OP}_{\mathtt{t}_1}\,\mathtt{l}_1.\cdots.\text{OP}_{\mathtt{t}_k}\,\mathtt{l}_k$. First, we need to show that $S'' = S_1'.\text{OP}_{\mathtt{t}}\,\mathtt{l}$ is safe. Suppose otherwise. Then, it must be the case that either (a) there exists some $\text{OP}_{\mathtt{t}'}\,\mathtt{l} \in S_1'$ such that $\mathtt{t} < \mathtt{t}'$, and $\text{OP}_{\mathtt{t}}\,\mathtt{l} = \mathbf{wt}_{\mathtt{t}}\,\mathtt{l}$, or (b) there exists a $\mathbf{wt}_{\mathtt{t}'}\,\mathtt{l} \in S_1'$ such that $\mathtt{t}' > \mathtt{t}$. If either of these conditions hold, however, S would not be

```
public interface Future<V> {
    V get()
      throws InterruptedException,
             ExecutionException;
}

public interface Callable<V> {
    V call() throws Exception;
}
```

```
public class FutureTask<V>
  implements Future<V>, Runnable {
    FutureTask(Callable<V> callable)
      throws NullPointerException
    { ... }
    V get()
      throws InterruptedException,
             ExecutionException
    { ... }
    void run() { ... }
}
```

Figure 5.4. The existing `java.util.concurrent` futures API

safe. Thus, by Lemma 1, we can permute $\text{OP}_{\mathtt{t}_k}\,\mathtt{l}_k$ with $\text{OP}_{\mathtt{t}}\,\mathtt{l}$ to yield a new safe schedule $S_p = S'''.\text{OP}_{\mathtt{t}_j}\,\mathtt{l}_j.\text{OP}_{\mathtt{t}}\,\mathtt{l}.\text{OP}_{\mathtt{t}_k}\,\mathtt{l}_k$. We can apply Lemma 1 again to $\text{OP}_{\mathtt{t}_j}\,\mathtt{l}_j.\text{OP}_{\mathtt{t}}\,\mathtt{l}$, and so on, repeatedly shifting $\text{OP}_{\mathtt{t}}\,\mathtt{l}$ until a serial schedule is constructed.

5.2 Design

Adding futures to Java raises several important design issues. Our foremost design goal is to preserve the spirit of programming with futures that made it so appropriate for functional programming: the expectation that a future, despite being executed asynchronously, performs its computation *as if* it had been invoked as a synchronous method call. We believe that strong notions of safety for futures is what makes them so powerful, where safety is ensured by the run-time system rather than left as a burden for the programmer.

We now proceed to discussion of an API for safe futures, their associated programming model, and their interaction with existing Java concurrency mechanisms. We will also describe the design of mechanisms used to preserve safety.

```
public class SafeFuture<V>
  implements Future<V>, Runnable {
    SafeFuture(Callable<V> callable)
      throws NullPointerException
    { ... }
    V get()
      throws InterruptedException,
             ExecutionException
    { ... }
    void run() { ... }
}
```

Figure 5.5. Safe futures API

5.2.1 API for Safe Futures

A major challenge in introducing any new language abstraction is to make it intuitive and easy to use. To ground our design, we begin with the existing Java futures API [34] that is now part of the Java 2 Platform Standard Edition 5.0 (J2SE 5.0). Snippets of this existing API appear in Figure 5.4, which embodies futures in the interface `Future`. The `get` operation (equivalent to the claim operation in the original formulation of futures) on a `Future` simply waits if necessary for the computation it encapsulates to complete, and then retrieves its result. We omit here those operations on futures that are not relevant to our remaining discussion.

In J2SE 5.0, an implementation of the `Future` interface is provided by the class `FutureTask`. Again, we omit details not relevant to our discussion. Here, the constructor for `FutureTask` creates a future that will, upon invocation of the `run` method, execute the given `Callable` by invoking its `call` method. If the call throws an exception, it is delivered to the caller at the point where it invokes the `get` method, wrapped up in an `ExecutionException`.

Our design calls for a new implementation of `Future`, namely `SafeFuture`, which is presented in Figure 5.5. Our semantics for `SafeFuture` demand that the program fragments appearing in Figure 5.6 be semantically equivalent, regardless of the computation

```
Callable<V> c = ...;

                    ⋮

...                 |  ...
V v = c.call();     |  Future<V> f
...                 |    = new SafeFuture<V>(c);
                    |  f.run();
                    |  ...
                    |  V v = f.get();
```

Figure 5.6. Semantically equivalent code fragments

performed by the given `Callable<V> c`, and the code surrounding its invocation, being performed as a simple call or as a future.

To preserve the transparency of future calls, any uncaught exception thrown by the future call (*i.e.*, from the `call` method of the `Callable`) will be delivered to the caller at the point of invocation of the `run` method, and the effects of the code following the run method will be revoked. The effects of the future call up to the point it threw the exception will remain. These semantics preserve equivalence with the simple call.

A more detailed example program appears in Figure 5.7. A future defined in the sample code fragment computes the sum of the elements in the array of integers `a` concurrently with a call to the static method `bar` on class `Foo`, which receives argument `a`. Note that method `bar` may access (and modify) `a` concurrently with the future computation. Our semantics require that the observable behavior of calls to methods `serial` and `concurrent` be the same. Replacing uses of `SafeFuture` with the existing `FutureTask` from J2SE 5.0 provides no such guarantee.

5.2.2 Programming Model

The programming model enabled by use of safe futures permits straightforward exploitation of latent parallelism in programs. One can think of safe futures as transparent annotations on method calls, which designate opportunities for concurrency. Serial pro-

```
public class Example implements Callable<Integer>
{
  int[] a = new int[]{1,2,3};

  public Integer call() {
    int sum = 0;
    for (int v : a) sum += v;
    return sum;
  }
  int serial() {
    Integer sum = call();
    Foo.bar(a);
    return sum;
  }

  public static void main (String[] args) {
    int serial = new Example().serial();
    int concurrent = new Example().concurrent();
    assert serial == concurrent;
  }
}
```

```
int concurrent() {
  Future<Integer> f
    = new SafeFuture<Integer>(this);
  f.run();
  Foo.bar(a);
  return f.get();
}
```

Figure 5.7. Using safe futures (with automatic boxing/unboxing of `int`/`Integer` supported by J2SE 5.0)

grams can be made concurrent simply by replacing standard method calls with future invocations. This greatly eases the task of the programmer, since all reasoning about the behavior of the program can be inferred from its original serial execution. Even though some parts of the program are executed concurrently, the semblance of serial execution is preserved. Of course, the cost of using futures may outweigh exploitable parallelism, so placement of future invocations has performance implications.

Under our current programming model, safety does not extend to covering the interaction between futures and Java threads. Threads which execute concurrently with futures might observe the actions of concurrently executing futures and their continuations out-of-order. Threads could be also incorrectly used to pass partial computation results between a future and its continuation thus violating serial execution semantics. Similarly, execution of operations with unpredictable side-effects, such as native method calls, is forbidden

when using futures. Before such an operation can be executed, all futures executing in the system must be fully evaluated and claimed.

5.2.3 Logical Serial Order

Our safety requirement demands that the observable behavior of a program using futures must be independent of whether futures are evaluated synchronously (serially within a single thread) or asynchronously (concurrently by multiple threads). The task of maintaining this *logical serial order* of operations in the presence of concurrent updates to shared state is non-trivial. Our solution is to encapsulate every fragment of computation that is fully evaluated within a single thread into an optimistic transaction. A transaction may thus encapsulate execution of either a future or its continuation. The serializability requirement is however not sufficient in this case to ensure safety because it only guarantees that transactions appear to execute in some serial order. Therefore we define total order over transactions (called *transaction order*) that represents the logical serial order and use this order to identify harmful data dependencies between operations of different transactions.

Conceptually, the execution of a program begins within a *primordial transaction* evaluated within the main thread of computation. Consider what happens when a future is scheduled for evaluation – *i.e.*, its `run` method is executed. Logically, the code fragment encapsulated within a future executes before the code fragment following the call to the `run` method up to the point where the future is claimed by the `get` operation (within the future's continuation). In order to preserve logical execution order, we create two more transactions: one associated with a thread used to evaluate the future – a *future transaction* executed within a freshly created thread; and one associated with the thread used to execute the future's continuation – a *continuation transaction* executed within the same thread as the primordial transaction. At this point we establish an execution order over these three transactions that reflects the logical serial order of execution in which the effects of the primordial transaction are visible to the future transaction whose effects are

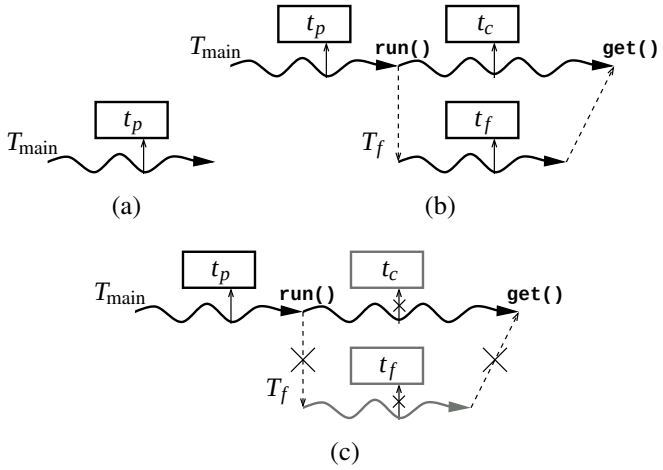


Figure 5.8. Transaction creation

in turn visible to the continuation transaction. Once evaluation of the future has successfully completed[1] and its transaction has been committed, execution of the continuation may also complete if it has already advanced to the point of the claim. If the execution of the continuation did not violate logical serial order, the result of the future's evaluation can be claimed (its respective `get` method invoked) and the continuation transaction can be committed. Then the execution can be returned to its original state (*i.e.*, all operations being performed within the main thread of computation). If the logical serial order has been violated, the continuation transaction must be aborted and re-executed. If evaluation of the continuation is completed (*i.e.*, the point of claiming the result of the future's evaluation is reached) before completion of the future's evaluation, termination of the continuation transaction is delayed until the future transaction is committed and the result of the future's evaluation is available to be claimed. Note that the future transaction and the continuation transaction are obliged to commit in compliance with the transaction order. Otherwise, if the continuation transaction was to be committed first, the remaining operations performed by the uncommitted future transaction might still be able to compromise the logical serial order.

[1]Note that in this situation the future transaction will always commit successfully. No violation of logical serial order between the future transaction and the primordial transaction can occur because the primordial transaction executed fully before the future transaction was been started.

As an example, consider Figure 5.8 illustrating execution of the `concurrent` method shown in Figure 5.7 (wavy lines represent threads and boxes represent transactions). Initially, only a primordial transaction (t_p) exists – it is bound to T_{main}, the thread evaluating the main method (Figure 5.8(a)). When a future is scheduled for execution (*i.e.*, its `run` method is invoked), two more transactions are created (Figure 5.8(b)): transaction t_f to evaluate the future (t_f is bound to T_f, a new thread used to execute the code encapsulated within the future), and transaction t_c to evaluate the continuation of the future (t_c is bound to the same thread as the primordial transaction t_p, in this case T_{main}). The execution of the program proceeds concurrently until the `get` method is invoked (the result computed by the future is then claimed) and then goes back to executing entirely within T_{main}, the main thread of computation. Note that at this point both the meta-data associated with transactions t_f and t_c as well as thread T_f could be discarded (Figure 5.8(c)) and cached for later re-use.

An ordering analogous to the one described above is created for all transactions created throughout the execution of a program. Consider a scenario when another future is scheduled for execution within an already existing continuation in Figure 5.8(b) (before method `get` is executed). Two more transactions must then be created: transaction t'_f to evaluate the new future and transaction t'_c to evaluate this future's continuation. Transaction t'_c will be executed by thread T_{main}, but a new thread T'_f will have to be created for the execution of the future transaction t'_f. The transaction order in which these transactions are allowed to attempt their commits (equivalent to the logical serial order they are obliged to maintain) is: t_f followed by t_c followed by t'_f followed by t'_c.

5.2.4 Preserving Serial Semantics

When two or more transactions execute concurrently, their operations may be arbitrarily interleaved and thus the semblance of serial execution may be violated. Consider two transactions: future transaction t_f and continuation transaction t_c. Under the logical serial

order of execution, t_f precedes t_c. If t_f and t_c execute concurrently, this order may be violated in one of two ways:

- t_c does not observe the effect of an operation performed by t_f (*e.g.*, a read in t_c does not see modification of shared data by t_f), even though it would have observed this effect if t_f and t_c were executed serially. We call this a *forward dependency violation*.[2]

- t_f does observe the effect of an operation performed by t_c that could never occur if t_f and t_c were executed serially because t_f would execute fully before t_c. We call this a *backward dependency violation*.

An example of schedules demonstrating both forward and backward dependency violations between transactions t_f and t_c, along with code snippets representing transactions, appear in Figure 5.9. In Figure 5.9(a) the continuation transaction t_c should see the result of the write to `o.foo` performed by the future transaction t_f. In Figure 5.9(b) the future transaction t_f should not see the result of the write to `o.bar` performed by the continuation transaction t_c. Note that the notion of a dependency violation captures the same properties as the schedule safety rules from Section 5.1.1 (forward dependency violations are captured by the *csafe* rule and backward dependency violations are captured by the *fsafe* rule).

The implementation of safe Java futures adapts our mechanisms used to support optimistic transactions (described in Section 2.4) to detect forward and backward dependency violations between transactions and to revoke transactions violating these dependencies in order to maintain the logical serial order of transactional execution.

5.3 Implementation

Our implementation prevents forward dependency violations by tracking dependencies between all transactional data accesses. Transactions violating forward data dependencies are aborted and automatically revoked – their effects are undone and transactions are

[2]Forward in the sense that an operation from the "logical future" causes the violation.

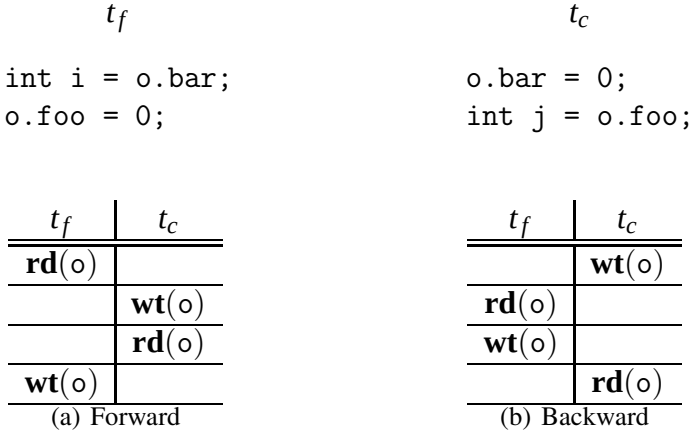


Figure 5.9. Dependency violations

restarted. Backward data dependencies are prevented by versioning items of shared state to ensure that each transaction updates only its private versions of shared items, preventing other transactions in its logical future from seeing the updates. Support for dependency tracking and versioning is provided using read and write barriers (as described in Section 2.4), inserted by the compilers available with the distribution of our implementation platform, Jikes RVM. A detailed description of these mechanisms is presented below.

5.3.1 Dependency Tracking

Dependencies among shared data accesses are tracked using access maps, as described in Section 2.3. Two maps are associated with every transaction: a *read map* to record reads and a *write map* to record updates. When transaction t is about to terminate, its read map is checked against the write maps of all transactions from t's logical past to determine if updates performed by these transactions might have caused forward dependency violations with respect to t's read operations. If intersection of t's read map with any of the write maps in non empty, transaction t must abort and be revoked. Otherwise, transaction t is allowed to commit.

We illustrate how our system handles forward dependency violations using the code fragment and sample schedule from Figure 5.9(a). We assume that future transaction t_f

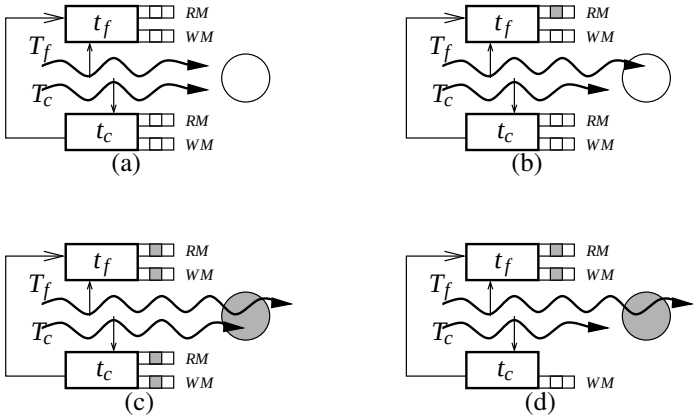


Figure 5.10. Handling of a forward dependency violation.

is executed by (and thus bound to) thread T_f and continuation transaction t_c by thread T_c. The entire scenario is illustrated in Figure 5.10, where wavy lines represent threads T_f and T_c, and a circle represents object o (it is marked gray when updated). Transactions are linked together in order to allow transactions from the logical future to access the maps of transactions from the logical past. There is a read map and write map associated with each transaction (each map has three slots and we assume that object o hashes to the second slot).

Execution starts with the future transaction reading a field of object o (Figure 5.10(a)) and tagging the appropriate slot in its read map. The continuation transaction then both reads and updates the same object, tagging its read map and write map appropriately (Figure 5.10(b)). Subsequently, the future transaction writes to the field of object o and tags its own write map (Figure 5.10(c)). At this point the future transaction gets committed (no dependency violations that could cause its revocation are possible since there were no other concurrent transactions executing in its logical past). However, before the continuation transaction can be committed, a check for forward dependency violations must be performed. This check fails since t_c's read map and t_f's write map overlap. The continuation transaction is revoked – its effects are undone and the transaction is re-executed (Figure 5.10(d)). Note that after revocation, no transactions in its logical past exist (the

future transaction has been committed). As a consequence, re-execution is guaranteed to succeed so maintaining its read map is unnecessary.

Since reads significantly outnumber writes in most Java programs, reducing the number of read barriers is critical to achieving reasonable performance. Our implementation therefore trades off accuracy for efficiency in detecting dependency violations. Instead of placing barriers on all read accesses to shared items (*e.g.*, reading an integer field from an object), we assume that once a reference is read from the heap, a transaction reading it will eventually read from the object targeted by that reference. Thus, the read barrier is placed only on loads of references from the heap (*e.g.*, `getfield` or `arrayload` bytecodes in which the type of the field or element is a reference). In other words, we "pre-read" all objects to which a transaction holds references (when a transaction is started we must apply the pre-read operation to all references in the current activation record). This optimization is applied only for objects and arrays to eliminate read barriers on them. All other accesses, including reads from static variables, and all writes to shared items incur the appropriate barrier.

Note that access maps are maintained only if there is more than one transaction executing in the system (*i.e.*, there is potential for concurrency and thus logical serial order violations). That is, barriers are responsible only for fetching the most recent version of an item if only the primordial transaction is active. Read and write map maintenance is in reality optimized even further: the first of a series of transactions does not need to record its reads because versioning ensures they cannot be compromised by any concurrent writes. Thus, it does not need to maintain a read map.

5.3.2 Revocation

The implementation of revocation for future transactions is simple. Because a future transaction is evaluated within a separate thread, we can simply terminate the thread and, after restoring local state,[3] re-start the execution of the future transaction. Revocation of

[3]No shared state is modified until transaction commit.

continuation transactions is implemented using a modified version of the exception handling mechanism described in Section 2.5. The exception handler for the `Revoke` exception wraps the scope of every method containing invocation of a future. Similarly to the implementation of revocable monitors (described in Section 4.2.2), we use BCEL framework to insert the exception handler and the code responsible for recording local state at the point when the continuation transaction starts. Another similarity with the implementation of revocable monitors is modification of the exception handling mechanism to suppress execution of default handlers during processing of the `Revoke` exception.

Our current implementation is unable to preserve state beyond the scope of a method containing invocation of a future (this would require the ability to preserve the full execution context of an arbitrary thread, including instruction pointer, registers, thread stack, *etc.*). Therefore, futures that are invoked but not claimed by the end of the method are implicitly claimed before the invoking method can return (*i.e.*, we wait for all futures to complete their execution), even though the matching `get` operation is still to be invoked.

5.3.3 Shared State Versioning

We use versioning of shared state to avoid backward data dependency violations and prevent updates of shared data from being made prematurely visible to other threads in case of revocations. The implementation of versioning is based on the general procedure described in Section 2.2.2 and employs lazy propagation of updates. Versioning is only used when more than one transaction is present in the system, since it is only then that concurrent shared data accesses may occur. Whenever a transaction attempts to write to an object, array, or static variable, the run-time system creates a private version of that item on which to perform the write. When a transaction gets committed, all versions created by this transaction become *committed versions* – they are designated to contain the most up-to-date values and used for all subsequent accesses. We handle object and array updates identically and use a similar procedure to handle updates to static variables. The code implementing the versioning procedure resides in the read and write barriers.

Object and Array Versioning

Because objects and arrays are treated identically, we refer only to objects when describing the versioning procedure. We extend the header of every object with a forwarding pointer. At object allocation time, this forwarding pointer is initialized to `null`. As the program executes, subsequent versions are appended to a circular list rooted at the forwarding pointer of the original object (*i.e.*, the original object is the head and tail of its version list). Each version is tagged with the unique identifier of the transaction that created it. This enables each transaction to locate its version in the list. The versions are sorted under the transaction order (described in Section 5.2.3).

We now describe the implementation of read and write operations on objects as performed by transactions in the presence of versioning. If only one transaction is present in the system (no concurrency), a read or write operation retrieves and accesses the most recent (committed) version of the object (or the original object in case no versions of it have been created). Otherwise, a more complicated access procedure is required.

Reads A reference to a shared object `o` referenced by transaction t must point to `o`'s most recent (committed) version with respect to t; in particular, t must not access any version of `o` that has been created by another transaction that occurs in t's logical future. To implement this invariant, we traverse the versions list (in transaction order) and either load the reference for the version tagged by t, or the version corresponding to t's most recent predecessor (according to transaction order). If transaction t' that occurs in t's logical past has written to `o`, and t reads the version corresponding to this write, a forward dependency violation exists and will be captured using the access maps, as described earlier. Indeed, the only instance when a read by t to a version written by t' would be safe is precisely when, at the point the read occurs, t' has already committed.

The implementation of read operations is additionally complicated by the fact that read barriers are only executed at reference loads. Thus, in order for loads of primitive values to proceed correctly, we maintain an invariant that no existing reference on the thread stack belonging to transaction t can point to a version created by any other transaction

executing in t's "logical" future. This invariant is relatively easy to maintain since the run-time system monitors all reference loads within read barriers. However, we must take special care to make sure that if a transaction creates a version, all references on the stack of the thread executing the transaction are updated correctly to point to this version (in other words, all reads performed by t must observe t's writes). We implement the invariant using a thread stack inspection mechanism described below.

Writes When multiple transactions are present in the system, all of them operate over their own local versions of shared data. In order to reduce the number of copies created, our implementation employs a copy-on-write strategy – a new version is created only when transaction t updates an object for the first time; we guarantee that all subsequent accesses by t will refer to that version.

All object update operations (including writes to primitive fields) are mediated by write barriers. When t performs an initial write to an object o, no local version of o exists. A new version is therefore created and inserted at the appropriate position in the version list rooted at o to reflect transaction order. At this point, other references to the same object may exist on t's thread stack. For example, t might have previously read o, but not yet written to it. All such references must then be forwarded to point to the freshly created version of o in order to avoid accessing stale versions. All versions of o created by transactions in t's logical past are considered stale with respect to t if t creates a new version of o.

Reference forwarding requires thread stack inspection as described below. Note that once the new version is created and all the references on the stack are forwarded, all the references on the stack throughout the entire execution of this transaction will always point to the right version (because subsequent reference loads are forwarded to the appropriate version). As a result, we avoid having to locate the correct version on the versions list when executing writes so long as a private copy exists. We only have to traverse the versions list upon version creation (when the object is first written). New versions are inserted at the appropriate place in the version list to maintain it in order.

All transactions maintain a list of their versions and use this list on abort to purge the revoked versions. Our implementation does not currently purge stale committed versions from an object's versions list. Instead, we defer such cleanup to the garbage collector.

Thread Stack Inspection We use a modified version of the thread stack inspection mechanism used by the garbage collector to support both pre-reading and forwarding of references on the stack. However, the essence of the mechanism remains the same. One of the major differences between the original stack inspection mechanism used during garbage collection and our modified version lies in the choice of the client using this mechanism. Garbage collection assumes that the stacks of inactive threads are being inspected. As a result, the entire execution state (including registers) of the inspected thread is available for inspection. In our system, the active thread inspects its own state. We artificially create a "snapshot" of the current thread's execution state, execute the stack inspection routine, and restore the execution state to the point before the inspection routine was invoked. This snapshot procedure is implemented in assembly. The stack inspection routine either tags a read map for every reference encountered (when pre-reading the stack) or forwards all references encountered to point to the correct version (when forwarding references during copy-on-write).

Versioning of Static Variables

In Jikes RVM static variables are stored in a global symbol table called the JTOC. Static variables are versioned similarly to objects. A copy-on-write strategy is used, with a versions list holding per-transaction versions of static variables. Because we must version static variables of both primitive and reference types, we introduce the notion of a *version container*: a small object that *boxes* a value of the static variable into an object that can be put on the versions list.

Upon initial write to a static variable by a transaction, a version container for the corresponding variable is created. The type of the slot in the JTOC representing this variable is then modified to indicate that its value has been copied to the list of version contain-

ers. For subsequent writes, a container created by the transaction must be retrieved and the value it contains updated. When reading the value of a static variable, the appropriate version container on the containers list must be located (similarly to retrieving an object version – it is either the container created by the current transaction or the one directly preceding its position in the containers list).

Indirections in the JTOC are lazily collapsed after all futures have been successfully evaluated and the program reverts to executing within a single (primordial) transaction. From this point on, only the most recent value of each static variable can ever be used.

5.4 Experimental Evaluation

Our experiments with safe futures for Java explore their performance on both standard benchmarks and a synthetic benchmark. In both cases, we use futures in a straightforward rewrite of initially sequential benchmark programs. The standard benchmarks are drawn from the Java Grande [54] benchmark suite. We choose a subset of naturally parallelizable benchmarks, namely `series`, `sparse`, `crypt` and `mc`.

The synthetic benchmark is intended to expose the performance of our implementation across a range of benchmark parameters, such as read/write ratio and degree of shared access. The synthetic benchmark is based on the OO7 object database benchmark suite [13], modified to use futures in a parallel traversal of the OO7 design database.

For all benchmarks, we also run their original sequential version on the *unmodified* Jikes RVM, and use this as a baseline to which we normalize for comparison with their future-enabled parallel versions.

5.4.1 Experimental Platform

Our implementation uses version 2.3.4+CVS (with 2005/06/23 13:35:22 UTC timestamp) of Jikes RVM for both the futures-enabled and the baseline (used for comparison) configurations. Jikes RVM is configured with the defaults for the Intel x86 platform, using the adaptive compiler framework.

We run each benchmark in its own invocation of Jikes RVM, repeating the benchmark six times in each invocation, and discarding the results of the first iteration, in which the benchmark classes are loaded and compiled, to elide the overheads of compilation. We report mean execution times, with 90% confidence intervals, to illustrate the degree of variation.

Our hardware platform is a 700MHz Intel Pentium III symmetric multi-processor (SMP) with 2GB of RAM running Linux kernel version 2.4.20-20.9smp (RedHat 9.0). Our parallel executions run four futures simultaneously on the SMP using four separate processors, though we note that such runs create multiple sets of four futures for each iteration of the benchmark, so a series of futures are created in each run.

5.4.2 Benchmarks

As mentioned earlier, we draw upon benchmarks from the Java Grande suite, as well as the OO7 synthetic design database benchmark. The former are representative of ideal candidate applications for parallelization using futures. The latter is less amenable to parallelization due to the density of the benchmark data structures and degree of sharing among them. Nevertheless, OO7 represents a benchmark in which meaningful parameters can be varied easily to demonstrate their impact on the performance of our futures implementation.

Java Grande

Each of the selected Java Grande benchmarks was chosen for being straightforwardly parallelizable. They each perform substantial computations over elements stored in Java arrays or in Java vectors, where access to the data structures is encoded into loops over the respective elements. We parallelized these benchmarks by substituting futures for subsets of the loop iterations similarly to the way these benchmarks have been parallelized for distributed execution via a Java message-passing interface (MPJ) [54]. For the benchmarks that use arrays, this rewriting also includes partitioning arrays into subarrays in order to

capture locality (such transformations were also used with MPJ), and because the conflict detection mechanisms described earlier function at per-array granularities, rather than for fragments of arrays. The `series` benchmark performs Fourier coefficients computation, `sparse` multiplies an unstructured sparse matrix stored in compressed-row format with a prescribed sparsity structure, `crypt` performs IDEA (International Data Encryption Algorithm) encryption and decryption and the `mc` benchmark is an implementation of Monte Carlo Simulation.

We believe benchmarks like these are prime candidates for parallelization using futures. Note, however, that even though they could be rewritten to use futures with only small changes to their source code, and straightforward partitioning of their data, rewriting the OO7 benchmark was even simpler – no data partitioning was required – and involved modifying only the top-level control loop (detailed below).

The OO7 Benchmark

The OO7 benchmark suite [13] provides a great deal of flexibility for benchmark parameters (*e.g.*, database structure, fractions of reads/writes to shared/private data). The multi-user OO7 benchmark [12] allows control over the amount of contention for access to shared data. By varying these parameters we are able to characterize the performance of safe futures over a mixed range of workloads.

Benchmark description The OO7 benchmark operates on a synthetic design database, consisting of a set of *composite parts*. Each composite part comprises a graph of *atomic parts*, and a `document` object containing a small amount of text. Each atomic part has a set of attributes (*i.e.*, fields), and is connected via a bi-directional association to several other atomic parts. The connections are implemented by interposing a separate connection object between each pair of connected atomic parts. Composite parts are arranged in an *assembly* hierarchy; each assembly is either made up of composite parts (a *base* assembly) or other assemblies (a *complex* assembly). Each assembly hierarchy is called a *module*, and has an associated *manual* object consisting of a large body of text.

Table 5.1
Component organization of the OO7 benchmark

Component	Number
Modules	$M+1$, for M futures
Assembly levels	7
Subassemblies per complex assembly	3
Composite parts per assembly	3
Composite parts per module	5000
Atomic parts per composite part	20
Connections per atomic part	3
Document size (bytes)	2000
Manual size (bytes)	100000

Our implementation of OO7 conforms to the specification of the standard OO7 database. Our traversals are a modified version of the multi-user OO7 traversals. A traversal chooses a single path through the assembly hierarchy and at the composite part level randomly chooses a fixed number of composite parts to visit (the number of composite parts to be visited during a single traversal is a configurable parameter). When the traversal reaches the composite part, it has two choices:

1. Do a *read-only* depth-first traversal of the atomic part subgraph associated with that composite part; or

2. Do a *read-write* depth-first traversal of the associated atomic part subgraph, swapping the *x* and *y* coordinates of each atomic part as it is visited.

Each traversal can be done beginning with either a *private* module or a *shared* module. The parameters of the workload control the mix of these four basic operations: read/write and private/shared. To foster some degree of interesting interleaving and contention in the case of concurrent execution, our traversals also take a parameter that allows extra overhead to be added to read operations to increase the time spent performing traversals.

Benchmark configuration Our results are all obtained with a OO7 database configured as in Table 5.1. The top-level execution of our sequential OO7 benchmark operates as

shown in Figure 5.11(a). It performs I benchmark iterations, each benchmark iteration comprises M sets of traversals in which the private module ranges from module 1 to module M, module $M+1$ is used as the shared module, and the parameter p controls the mix of operations performed by the traversals.

The top-level execution of our futures-enabled OO7 benchmark operates as shown in Figure 5.11(b). It performs I benchmark iterations, each benchmark iteration comprises M futures, each of which performs a set of traversals operating on a distinct private module m, module $M+1$ is used as the shared module, and the parameter p controls the mix of operations performed by the traversals.

We seed the traversals with the same random seed in both the sequential and futures-enabled executions of the benchmark, such that both versions perform identical workloads.

5.4.3 Results

We present results for the Java Grande benchmarks first, to indicate the behavior of futures under ideal circumstances. OO7 is more demanding, but also more tunable, revealing the underlying performance characteristics of our implementation.

Java Grande results

Figure 5.12 reports the elapsed time for execution of the future-enabled versions of the Java Grande benchmarks, normalized against the average elapsed time for execution of

```
for (i = 1; i <= I; i++)
  for (m = 1; m <= M; m++)
    traversals(m,p);
```

(a) Sequential OO7 benchmark

```
for (i = 1; i <= I; i++) {
  for (m = 1; m <= M; m++)
    f[m] = future(traversals(m,p));
  for (m = 1; m <= M; m++)
    f[m].get();
}
```

(b) Parallel OO7 benchmark

Figure 5.11. Top-level loop of the OO7 benchmark

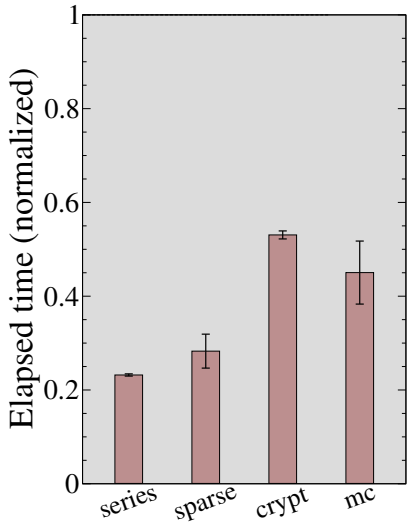
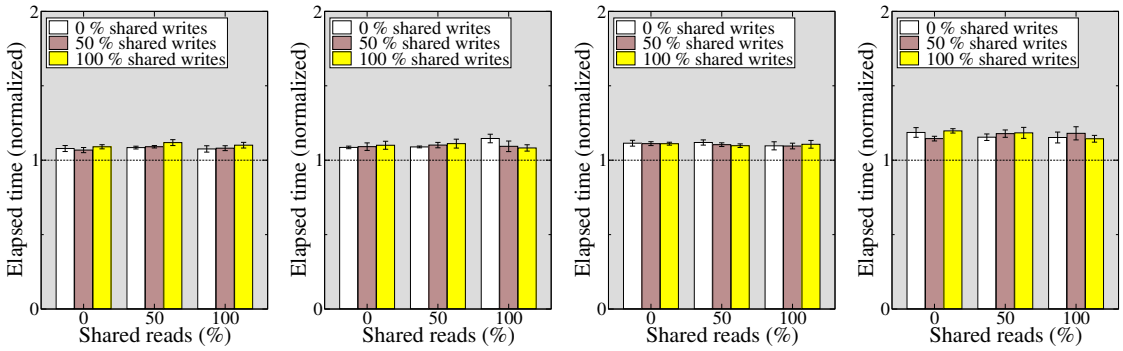


Figure 5.12. Java Grande: elapsed time (normalized)

the unmodified sequential benchmarks running on the unmodified Jikes RVM. Times are arithmetic means of the 5 hot runs of each benchmark, plotted with 90% confidence intervals. Recall that we parallelize the benchmarks using four futures running concurrently on four CPUs. Thus, observe that speedups range from perfect (or even slightly super-linear – 4× for `series`), to a little less than 2× speedup for `crypt`. We believe that the reason for the super-linear speedup for `series` is due to improved locality as a result of the array partitioning.

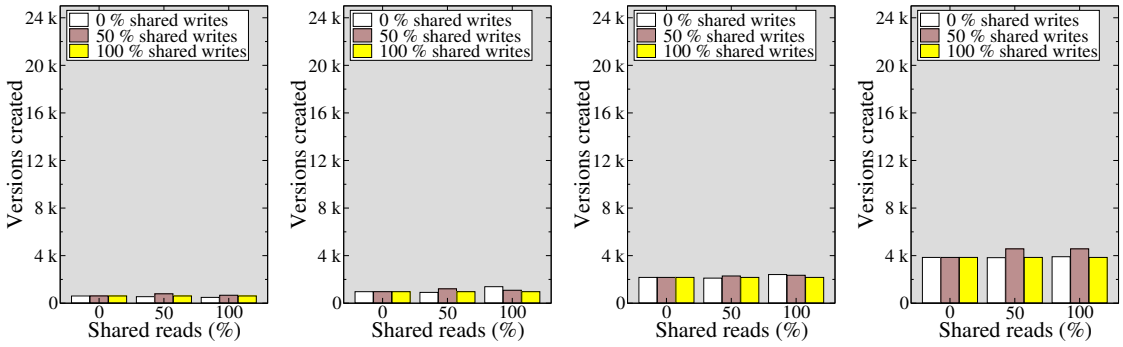
OO7 results

We report results for two basic versions of OO7, one for a database containing only 2 ($M = 1$) modules and one for a database comprising 5 ($M = 4$) modules. Again, we compare the future-enabled parallel versions against the sequential version of the benchmark. We vary the ratio of writes to reads performed within each set of traversals as 4%, 8%, 16% and 32% writes (96%, 92%, 84%, and 68% reads, respectively), in an attempt to model workloads with mutation rates ranging from low to moderate. We also vary the ratio of shared/private accesses for each mix of reads/writes as 0%, 50% and 100%. Thus,



(a) 4% writes, 96% reads (b) 8% writes, 92% reads (c) 16% writes, 84% reads (d) 32% writes, 68% reads

Figure 5.13. OO7 with 1 future: average elapsed time per iteration (normalized)



(a) 4% writes, 96% reads (b) 8% writes, 92% reads (c) 16% writes, 84% reads (d) 32% writes, 68% reads

Figure 5.14. OO7 with 1 future: versions created per iteration

for 4% writes, 50% shared, a set of 100 traversals will on average perform 2 read-write traversals to shared data, 2 read-write traversals to private data, 48 read-only traversals on shared data, and 48 read-only traversals on private data.

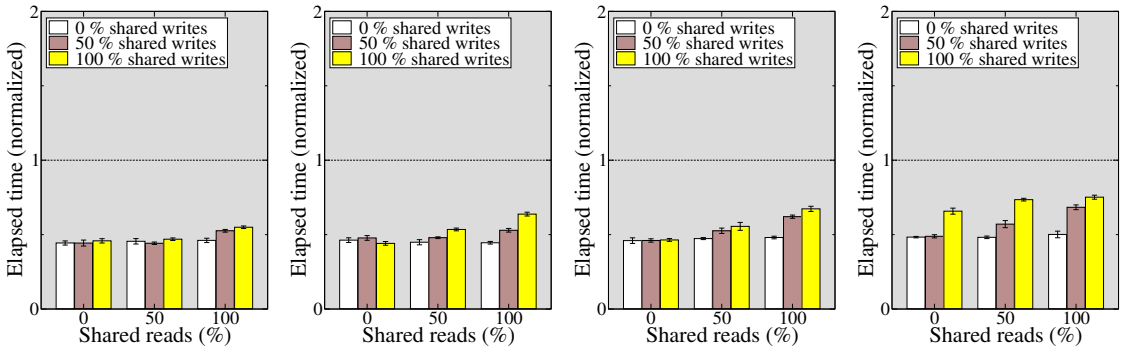
With just 2 ($M = 1$) modules, both the original and future-enabled versions are inherently sequential, since the degree of future-enabled parallelism is equal to M for a database containing $M + 1$ modules. Moreover because only one future is ever active, revocation cannot occur. Thus, the comparison for $M = 1$ yields a measure of the fundamental overheads in our system for creating and claiming futures (and indirectly the effectiveness of our context-caching mechanisms), for the read and write barriers used to track accesses, and for versioning. The elapsed time results, normalized against the sequential version running on the unmodified Jikes RVM, are presented in Figure 5.13. These reveal a per-

future performance hit of 8-12% for 4% writes. As write ratios increase, we see overheads of 15-20% for the 32% write ratio. Figure 5.14 graphs the number of versions created per benchmark iteration, showing that the number of versions created increases with sharing and the write ratio.

Of course, for more futures, this performance hit may come to dominate. Some of the overhead results from the lack of efficient support in Jikes RVM for caching of thread state (*e.g.*, stacks) from one thread activation to another. Thus, spawning a future is relatively expensive. Still, our overheads are low enough to justify the use of safe futures for a range of applications, as the Java Grande results illustrate.

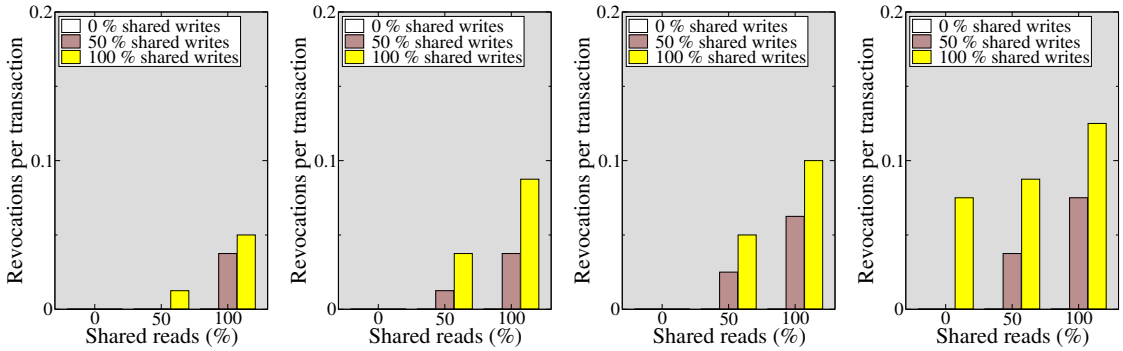
Adding concurrency yields opportunity for parallelism, as illustrated in the results for OO7 using four futures, shown in Figure 5.15. With four futures executing in parallel on four CPUs there is the possibility of revocation, which we graph in Figure 5.16. Without sharing there are no revocations. Thus for the unshared executions we see uniform gains of 52-56% across the range of write ratios, as expected. The performance gains vary depending on the configuration; even at 32% write ratio with 100% sharing we still observe a performance benefit of about 25% (Figure 5.15(d)). In all configurations, the revocations seem to impact performance significantly, since their rise is correlated with increased sharing, as well as write ratio (see Figure 5.16). The increase in versions created (Figure 5.17) also affects execution times – as write ratios increase, elapsed times in Figure 5.15 also increase slightly even for configurations where no revocations are observed.

The cost of creating versions constitutes part of the "base" overhead common across all configurations, though clearly non-existent in the sequential version of the benchmark. Another large base overhead results from executing large numbers of read barriers. We observe on average 63 million read barriers (30 million for objects, 18 million for arrays and 15 million for static variables) per benchmark iteration (these numbers remain much the same across all configurations). This indicates that our initial decision to minimize the number of barriers by inserting them only at reference loads was prescient. We also observe a large number of write barriers – 16 million on average per benchmark iteration (6.5 million for objects, 9.5 million for arrays, and a negligible number for static variables).



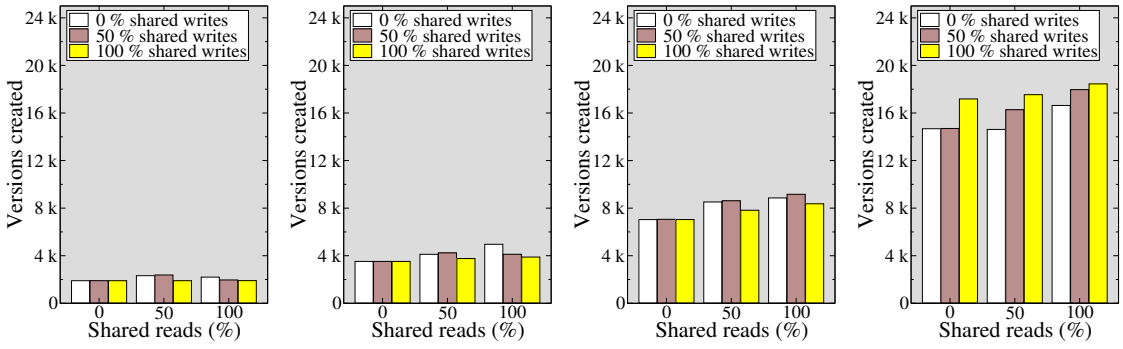
(a) 4% writes, 96% reads (b) 8% writes, 92% reads (c) 16% writes, 84% reads (d) 32% writes, 68% reads

Figure 5.15. OO7 with four futures: average elapsed time per iteration (normalized)



(a) 4% writes, 96% reads (b) 8% writes, 92% reads (c) 16% writes, 84% reads (d) 32% writes, 68% reads

Figure 5.16. OO7 with four futures: revocations per iteration



(a) 4% writes, 96% reads (b) 8% writes, 92% reads (c) 16% writes, 84% reads (d) 32% writes, 68% reads

Figure 5.17. OO7 with four futures: versions created per iteration

The number of write barriers for objects increases as the number of writes to shared objects grows across different configurations. We are particularly penalized by the number

of static variable accesses for this implementation of OO7, which uses them to capture the traversal parameters. In general, static analyses such as escape analysis could be very helpful in optimizing away unnecessary barrier overheads [9, 10, 14]. We note, however, that for OO7 such analyses are unlikely to have much impact, because all futures operate over a single recursively-defined data structure. Nonetheless, even without the benefit of advanced compiler optimizations, the performance of our implementation using just run-time optimizations is encouraging.

5.5 Related Work

The semantics of futures [18, 19] and their implementation [36, 44] have been well-studied in the context of functional languages. However, from the point of view of the work presented in this thesis, the most interesting research efforts in this area concern application of futures in the context of imperative programming languages.

Promises [41] are a variant of futures for a statically typed language, Argus [40]. They are used to implement asynchronous remote method calls. Futures, in their original interpretation, are expression annotations that may or may not be taken into account by the run-time system. As a result, the result of their evaluation may be either the placeholder for the actual value (if the run-time system evaluates the expression asynchronously) or the value itself (if evaluation is synchronous). Promises are strongly typed and the operation of claiming a promise is made explicit, which avoids any run-time check to distinguish the different types of returned values. Making the claim operation explicit also allows convenient handling of run-time exceptions and problems related to the distributed setting (*e.g.*, node failures).

More recently, Pratikakis *et al.* [48] present a static analysis to allow Java programs to use futures without requiring wholesale changes to the program to satisfy type restrictions. Their analysis tracks how an object representing a future flows through a program, and injects coercions that perform a claim operation on the object at points where the value yielded by the future, rather than the object representing the future, is required. The anal-

ysis uses qualifier inference to track how futures are used. Our goals are similar in spirit to their work in that both attempt to treat futures as a transparent concurrency mechanism. However, unlike our design and implementation, Pratikakis *et al.* make no guarantees that the evaluation of a future does not introduce behavior inconsistent with the sequential program from which it was derived. Although we expect that futures are primarily useful for spawning concurrent tasks that exhibit relatively little to modest sharing, it is nonetheless critical that safety violations be detected when they do occur.

Safe futures are a mechanism allowing relatively straightforward parallelization of sequential Java programs. The ParaTran project [35] was an attempt to achieve similar goals for sequential Lisp programs in the presence of side-effects. A sequential Lisp program is divided into tasks that can be executed concurrently, using a compile-time analysis. Like our implementation of safe futures, ParaTran uses optimistic concurrency control techniques to monitor data accesses performed by concurrently executing tasks and to revoke fragments of computation after detecting violations of the (logical) serial execution order. To the best of our knowledge however, ParaTran has never been implemented to run on a real system and the available simulation results do not include all the potential costs of such an implementation. One of the major goals of our work was a thorough performance evaluation of an implementation based on a realistic language execution environment.

Another approach to parallelizing sequential programs in the presence of side-effects has been explored in the context of the Jade programming language [50]. Jade is a high-level, implicitly parallel language designed to exploit coarse-grained concurrency. It has been implemented on a wide variety of platforms, ranging from shared memory multiprocessor machines to loosely coupled networks of workstations using message-passing. It has been proven to be effective in parallelizing sequential programs (up to linear speedups have been achieved).

Jade provides a programmer with an abstraction of both single address space and serial semantics. In order to parallelize a sequential program, the programmer must delimit code fragments (tasks) that can be executed concurrently and explicitly specify invariants describing how different tasks access shared data. The run-time system is then responsi-

ble for exploiting available concurrency and verifying data access invariants in order to preserve the semantics of the serial program. Violations of data access invariants result in run-time errors.

The most recent approach to automatic parallelization of sequential Java programs has been developed by Garcia *et al.* [49]. Their Mitosis compiler enables automatic extraction of thread-level parallelism through speculative execution of threads. Their system estimates, based on the cost-benefit model, whether spawning of a new speculative thread has the potential to improve overall run-time performance. Spawning of a thread consists of two separate operations: a *spawning point* (*SP*) and a *control quasi-independent point* (*CQIP*). The SP identifies the point where speculative thread is created and the CQIP identifies the point when speculative thread starts executing. Correctness of the thread's execution after the CQIP relies on the processors state (memory and register values) at the CQIP being correctly predicted. This prediction is encapsulated withing a *pre-computation slice* (*p-slice*), computed by the compiler. Mispredictions are handled by the existing hardware. Performance evaluation of the Mitosis compiler architecture (used to parallelize several sequential benchmarks from the Olden suite) shows an average speedup of over 100%.

5.6 Conclusions

In this chapter, we have described how optimistic transactions can be used to support safe futures in Java. Futures provide a simple and intuitive API for concurrent programming that allows a concurrent program to be constructed often through only a small rewrite of a sequential one. Unfortunately, futures as currently specified in Java are not treated as a semantically transparent annotation, thus significantly weakening their utility. Programmers who use futures must reason about the subtle interactions among future-encapsulated computations, in much the same way they must reason about the interaction of threads in a typical multi-threaded Java program. Safe futures obviate the need for such reasoning by guaranteeing that their injection into a sequential program does not alter the observable behavior of the program. Furthermore, the cost of providing this added level of safety is

not prohibitive. The evaluation of our implementation indicates that safe futures can be used to exploit concurrency even for applications with modest mutation rates on shared data.

6 TRANSACTIONAL MONITORS

Programmers developing concurrent applications often reason about safety of concurrent execution in terms of such high level properties as *isolation* and *atomicity*. It has been widely recognized that it is difficult to express and enforce these properties using low-level mechanisms such as mutual exclusion synchronization. We have discussed the most common problems related to using mutual exclusion in Section 1.1.

Recent proposals recognize that such high level properties can be enforced by concurrency control mechanisms that avoid the problems of locking, by transplanting notions of transactions to the programming language context [27, 30, 56]. These mechanisms ensure atomicity and isolation of operations performed within a transaction, while enhancing concurrency by permitting the operations of different transactions to be interleaved as long as the resulting schedule is serializable. Atomicity is a powerful abstraction, permitting programmers to more easily reason about the effects of concurrent programs independently of arbitrary interleavings. There is comprehensive empirical evidence that programmers almost always use mutual-exclusion locks to enforce properties of atomicity and isolation [20]. 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 modifying a single memory word using an atomic compare-and-swap operation. 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 commit. 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 accrue only when contention would otherwise impede concurrency and serializability violations are low.

These trade-offs argue for consideration of a hybrid approach, where existing concurrency abstractions used to enforce atomicity and isolation, such as Java's monitors, can be implemented by either locks or transactions. In fact, from a programmer's perspective it is irrelevant whether threads entering a monitor acquire a mutual-exclusion lock or execute transactionally, so long as the language-defined properties of the monitor are enforced. Dynamically choosing which style of execution to use based on the observed contention for the monitor permits the best of both worlds: low-cost locking when contention is low, and improved concurrency using transactions when multiple threads attempt to simultaneously execute within the monitor.

Complicating this situation is the issue of nesting, which poses both semantic and implementation difficulties. The closed nested transaction model [45] represents the purest expression of nested transactions for preserving atomicity and isolation. In this model, when an inner transaction commits, isolation semantics for transactions mandate that its effects are not globally visible until the outermost transaction in which it runs successfully commits. In contrast, Java monitors expressed as synchronized methods/blocks reveal all prior effects upon exit, even if the synchronized execution is nested inside another monitor. Obtaining a meaningful reconciliation of locks with transactions requires addressing this issue.

We now proceed to describing how locks and transactions can be reconciled within Java's monitor abstraction. Our treatment is *transparent* to applications: programs continue to use the standard Java synchronization primitives to express the usual constraints on concurrent executions. A synchronized block[1] may be guarded by a *transactional monitor* (implemented using transactional machinery) even if it was previously guarded by an *exclusive monitor* (implemented using mutual exclusion), and vice versa. Transactional execution dynamically toggles back to mutual exclusion whenever continuing

[1] A synchronized method can be expressed as a non-synchronized method whose entire body is enclosed in a synchronized block.

transactional execution becomes infeasible, such as at native method calls, whose effects, in general, cannot be undone. In both cases, hybrid execution does not violate Java semantics, and serves only to improve performance. We argue correctness of our approach using a formal semantics. We also describe design and implementation of a Java run-time supporting hybrid-mode execution and present a detailed implementation study that quantifies overheads inherent with our approach.

We now proceed to describe a formal semantics that defines safety criteria under which exclusive monitors and transactional monitors can co-exist. We show that for programs that obey standard atomicity properties, Java monitors can be realized using either of the concurrency control protocols with no change in observable behavior. In this way, we resolve the apparent mismatch in the visibility of the effects of Java monitors versus closed nested transactions.

6.1 Semantics

To examine notions of safety with respect to transactions and mutual exclusion, we define a two-tiered semantics for a simple dynamically-typed call-by value object calculus similar to Classic Java [22] extended with threads and synchronization. The first tier describes how programs written in this language are evaluated to yield a schedule that defines a sequence of possible thread interleavings, and a memory model that reflects how and when updates to shared data performed by one thread are reflected in another. The second tier defines constraints used to determine whether a schedule is safe based on a specific interpretation of what it means to protect access to shared data; this tier thus captures the behavior of specific concurrency control mechanisms.

The syntax of the language is presented in Figure 6.1. We take metavariables L to range over class declarations, `C` to range over class names, `t` to denote thread identifiers, `M` to range over methods, `m` to range over method names, `f` and `x` to range over fields and parameters, respectively, `l` to range over locations, and `v` to range over values. We use P for process terms, and `e` for expressions.

$$
\begin{array}{rcl}
P & ::= & (P \mid P) \mid \mathtt{t}[\mathtt{e}] \\
L & ::= & \mathtt{class}\ \mathtt{C}\{\overline{\mathtt{f}}\ \overline{\mathtt{M}}\} \\
\mathtt{M} & ::= & \mathtt{m}\,(\overline{\mathtt{x}})\,\{\mathtt{e}\} \\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{l} \mid \mathtt{this} \mid \mathtt{e.f} \mid \mathtt{e.f} := \mathtt{e} \mid \mathtt{e.m}\,(\overline{\mathtt{e}}) \\
 & \mid & \langle \mathtt{e} \rangle \mathtt{e} \mid \mathtt{new}\ \mathtt{C}\,() \mid \mathtt{spawn}\,(\mathtt{e}) \mid \mathtt{guard}\,(\mathtt{e})\,\{\mathtt{e}\}
\end{array}
$$

Figure 6.1. Language syntax.

A program defines a collection of class definitions, and a collection of processes. Classes are all uniquely named, and define a collection of instance fields and instance methods which operate over these fields. Every method consists of an expression whose value is returned as the result of a call to that method. Every class has a unique (nullary) constructor to initialize object fields. Expressions can read the contents of a field, store a new value into an instance field, create a new object, perform a method call, enforce sequencing of actions, or guard the evaluation of a subexpression.

To evaluate a guard expression of the form, $\mathtt{guard}\ (\mathtt{e})\ \{\mathtt{e}'\}$, expression $\mathtt{e}$ is first evaluated to yield a location $\mathtt{l}$. We refer to $\mathtt{l}$ as a *monitor* and use it to mediate the execution of the guarded expression $\mathtt{e}'$. Note, that evaluation of different expressions may be mediated by the same monitor. If only one thread at a time is allowed to evaluate any of the expressions guarded by the same monitor, the monitor acts as a mutual-exclusion lock (becomes an *exclusive monitor*). Otherwise, the monitor becomes a *transactional monitor* and is used to mediate execution of multiple threads by enforcing serializability of their actions. In the remaining part of this section we will discuss how these two types of monitors can transparently co-exist within the same framework.

The semantics of the calculus is presented in Figure 6.2 and Figure 6.3. In the following, we use over-bar to represent a finite ordered sequence, for instance, $\overline{\mathtt{f}}$ represents $\mathtt{f}_1\,\mathtt{f}_2\,\ldots\,\mathtt{f}_n$. The term $\overline{\alpha}\alpha$ denotes the extension of the sequence $\overline{\alpha}$ with a single element α, and $\overline{\alpha}\overline{\alpha'}$ denotes sequence concatenation, $S.\mathrm{OP}_{\mathtt{t}}$ denotes the extension of schedule S with operation $\mathrm{OP}_{\mathtt{t}}$. Given schedules S and S', we write $S \preceq S'$ if S is a subsequence of S'.

A value is either the distinguished symbol `null`, a location, or an object $\mathtt{C}(\overline{\mathtt{l}})$ (an instance of class `C`, in which field $\mathtt{f}_i$ has value $\mathtt{l}_i$). Program evaluation and schedule construction is specified by a reduction relation, $P, \Delta, \Gamma, S \Longrightarrow P', \Delta', \Gamma', S'$ that maps program states to new program states. A state consists of a collection of evaluating processes (P), a thread store (Δ) that maps threads to their local caches (represented by σ), and a global store (Γ). This last element of the program state requires additional explanation. As described below, updates performed by one thread become visible to other threads as a result of `guard` expressions being evaluated. The global store Γ is parametrized by locations representing monitors in order to allow different threads to observe different values residing in the same locations, depending on which `guard` expressions have been evaluated by these threads. Informally, threads evaluate expressions using their local caches, loading and flushing their caches at synchronization points defined by `guard` expressions. This semantics roughly corresponds to a release consistency memory model similar to the Java Memory Model [43], described in Section 4.1.2. An auxiliary relation $\rightarrow_{\mathtt{t}}$ is used to describe reduction steps performed by a specific thread `t` using its own local cache σ. Actions that are recorded by a schedule are those that read and write locations, and those that acquire and release monitors, the latter generated as part of `guard` expression evaluation.

In global evaluation rules $\mathtt{E}^{\mathtt{t}}_{P}[\mathtt{e}]$ denotes a collection of processes containing a process with thread identifier `t` executing expression `e` with context `E`. The expression "picked" for evaluation is determined by the structure of evaluation contexts. Most of the rules are standard: holes in contexts can be replaced by the value of the expression substituted for the hole. Method invocation binds the variable `this` to the current receiver object, in addition to binding actuals to parameters, and evaluates the method body in this augmented environment. Read and write operations augment the schedule in the obvious way. Constructor application returns a reference to a new object whose fields are initialized to `null`.

In order to spawn a new thread for evaluation of expression `e`, we first associate the new thread with a fresh thread identifier, set the thread's local cache to be the current local cache of its parent, and begin evaluation of `e` using an empty context.

Evaluation contexts: **Program states:**

$$
\begin{array}{rcl}
\mathtt{E} & ::= & \bullet \\
& | & \mathtt{E}[\bullet].\mathtt{f} := \mathtt{e} \\
& | & \mathtt{l}.\mathtt{f} := \mathtt{E}[\bullet] \\
& | & \mathtt{E}[\bullet].\mathtt{m}(\overline{e}) \\
& | & \mathtt{l}.\mathtt{m}(\overline{\mathtt{l}}\ \mathtt{E}[\bullet]\ \overline{\mathtt{e}}) \\
& | & \mathtt{E}[\bullet]\ ;\ \mathtt{e} \\
& | & \mathtt{guard}\ (\mathtt{E}[\bullet])\ \{\mathtt{e}\}
\end{array}
$$

$$\mathtt{E}^{\mathtt{t}}_{P}[\mathtt{e}] ::= P \mid \mathtt{t}[\mathtt{E}[\mathtt{e}]]$$

$$
\begin{array}{rcll}
\mathtt{t} & \in & \textit{Tid} & \\
P & \in & \textit{Process} & \\
\mathtt{x} & \in & \textit{Var} & \\
\mathtt{l} & \in & \textit{Loc} & \\
\mathtt{v} & \in & \textit{Val} & = \ \mathtt{null} \mid \mathtt{C}(\overline{\mathtt{l}}) \mid \mathtt{l} \\
\sigma & \in & \textit{Store} & = \ \textit{Loc} \rightarrow \textit{Val} \\
\Delta & \in & \textit{TStore} & = \ \textit{Tid} \rightarrow \textit{Store} \\
\Gamma & \in & \textit{SMap} & = \ \textit{Loc} \rightarrow \textit{Store} \\
\mathrm{OP}_{\mathtt{t}}\,\mathtt{l} & \in & \textit{Ops} & = \ \{\mathbf{rd}, \mathbf{wt}\} \times \textit{Tid} \times \textit{Loc} \\
\mathrm{OP}^{\Gamma}_{\mathtt{t}}\,\mathtt{l} & & & \{\mathbf{acq}, \mathbf{rel}\} \times \textit{Tid} \times \textit{Loc} \times \textit{SMap} \\
S & \in & \textit{Schedule} & = \ \overline{\mathrm{OP}_{\mathtt{t}}\,\mathtt{l} \mid \mathrm{OP}^{\Gamma}_{\mathtt{t}}\,\mathtt{l}} \\
\Lambda & \in & \textit{State} & = \ \textit{Process} \times \textit{Store} \times \textit{Schedule}
\end{array}
$$

Figure 6.2. Program states and evaluation contexts.

Let guard (l) {e} be an expression where evaluation of the guarded expression e is mediated using monitor l. Before evaluating e, local cache of the thread evaluating guard expression is updated to load the current contents of the global store at location l. In other words, global memory is indexed by the set of locations that act as monitors: whenever a thread attempts to synchronize against one of these monitors (say, l), the thread augments its local cache with the store associated with l in the global store. The guarded expression e is then evaluated with respect to this updated cache. When the evaluation completes, the converse operation is performed: the contents of the local cache is flushed to the global store indexed by l. Thus, threads that synchronize on different monitors will not have their updates made visible to one another. To simplify the presentation, we prohibit nested guard expressions from synchronizing on the same reference ($\mathtt{l} \notin \textit{lockset}(S, \mathtt{t})$, where $\textit{lockset}(S, \mathtt{t})$ represents a set of monitors acquired by t in S at the point of the guard expression evaluation).

Sequential evaluation rules:

$$\frac{\begin{array}{c}\mathtt{l}', \overline{\mathtt{l}} \textit{ fresh} \\ \sigma' = \sigma[\mathtt{l}' \mapsto \mathtt{C}(\overline{\mathtt{l}}), \overline{\mathtt{l}} \mapsto \mathtt{null}] \\ S' = S.\mathbf{wt}_{\mathtt{t}}\,\mathtt{l}_1.\ \ldots \mathbf{wt}_t\,\mathtt{l}_n.\mathbf{wt}_t\,\mathtt{l}' \\ \mathtt{l}_1, \ldots, \mathtt{l}_n \in \overline{\mathtt{l}}\end{array}}{\mathtt{new\ C}\ (), \sigma, S \rightarrow_{\mathtt{t}} \mathtt{l}', \sigma', S'} \qquad \frac{\begin{array}{c}\sigma(\mathtt{l}) = \mathtt{C}(\overline{\mathtt{l}''}) \qquad \sigma(\mathtt{l}') = \mathtt{v} \\ \sigma' = \sigma[\mathtt{l}''_i \mapsto \mathtt{v}] \\ S' = S.\mathbf{rd}_{\mathtt{t}}\,\mathtt{l}'.\mathbf{wt}_{\mathtt{t}}\,\mathtt{l}''_i\end{array}}{\mathtt{l.f}_i := \mathtt{l}', \sigma, S \rightarrow_{\mathtt{t}} \mathtt{l}', \sigma', S'}$$

$$\frac{}{\langle \mathtt{l} \rangle \mathtt{e}, \sigma, S \rightarrow_{\mathtt{t}} \mathtt{e}, \sigma, S} \qquad \frac{\begin{array}{c}\mathtt{class\ C}\,\{\overline{\mathtt{f}}\ \overline{\mathtt{M}}\} \in L \qquad \sigma(\mathtt{l}) = \mathtt{C}(\overline{\mathtt{l}'}) \\ S' = S.\mathbf{rd}_{\mathtt{t}}\,\mathtt{l}'_i\end{array}}{\mathtt{l.f}_i, \sigma, S \rightarrow_{\mathtt{t}} \mathtt{l}'_i, \sigma, S'}$$

$$\frac{\begin{array}{c}\sigma(\mathtt{l}) = \mathtt{C}(\overline{\mathtt{l}'}) = \mathtt{v} \qquad \sigma(\overline{\mathtt{l}}) = \overline{\mathtt{v}'} \\ \mathtt{class\ C}\,\{\overline{\mathtt{f}}\ \overline{\mathtt{M}}\} \in L \qquad \mathtt{m}\ (\overline{\mathtt{x}})\,\{\mathtt{e}\} \in \overline{\mathtt{M}}\end{array}}{\mathtt{l.m}\ (\overline{\mathtt{l}}), \sigma, S \rightarrow_{\mathtt{t}} [{}^{\mathtt{v}}/_{\mathtt{this}}, {}^{\overline{\mathtt{v}'}}/_{\overline{\mathtt{x}}}]\mathtt{e}, \sigma, S}$$

Global evaluation rules:

$$\frac{\begin{array}{c}\Delta(\mathtt{t}) = \sigma \\ \mathtt{e}, \sigma, S \rightarrow_{\mathtt{t}} \mathtt{e}', \sigma', S'\end{array}}{\mathtt{E}^{\mathtt{t}}_P[\mathtt{e}], \Delta, \Gamma, S \Longrightarrow \mathtt{E}^{\mathtt{t}}_P[\mathtt{e}'], \Delta[\mathtt{t} \mapsto \sigma'], \Gamma, S'}$$

$$\frac{\begin{array}{c}\mathtt{t}' \textit{ fresh} \qquad \Delta' = \Delta[\mathtt{t}' \mapsto \Delta(\mathtt{t})] \\ P' = P \mid \mathtt{t}'[\mathtt{e}]\end{array}}{\mathtt{E}^{\mathtt{t}}_P[\mathtt{spawn}\ (\mathtt{e})], \Delta, \Gamma, S \Longrightarrow \mathtt{E}^{\mathtt{t}}_{P'}[\mathtt{null}], \Delta', \Gamma, S}$$

$$\frac{\begin{array}{c}\Delta(\mathtt{t}) = \sigma \qquad \sigma' = \sigma \circ \Gamma(\mathtt{l}) \\ \Delta' = \Delta[\mathtt{t} \mapsto \sigma'] \\ \mathtt{l} \notin \textit{lockset}(S, \mathtt{t}) \\ P \mid \mathtt{t}[\mathtt{e}], \Delta', \Gamma, \phi \Longrightarrow^* P' \mid \mathtt{t}[\mathtt{l}'], \Delta'', \Gamma', S' \\ \Gamma'' = \Gamma'[\mathtt{l} \mapsto \Delta''(\mathtt{t})]\end{array}}{\mathtt{E}^{\mathtt{t}}_P[\mathtt{guard}\ (\mathtt{l})\ \{\mathtt{e}\}], \Delta, \Gamma, S \Longrightarrow \mathtt{E}^{\mathtt{t}}_{P'}[\mathtt{l}'], \Delta'', \Gamma'', S.\mathbf{acq}^{\Gamma}_{\mathtt{t}}\,\mathtt{l}.S'.\mathbf{rel}^{\Gamma'}_{\mathtt{t}}\,\mathtt{l}}$$

Figure 6.3. Language semantics.

6.1.1 Safety

Evaluation of an expression of the form `guard (l) {e}` by thread `t` results in a schedule being augmented to record the fact that the evaluation of guarded expression `e` has

been mediated by monitor $\mathtt{l}$. When evaluation of the `guard` expression starts, an acquire operation ($\mathbf{acq}_{\mathtt{t}}^{\Gamma}\,\mathtt{l}$) is inserted into the schedule. When the evaluation completes, a release operation ($\mathbf{rel}_{\mathtt{t}}^{\Gamma}\,\mathtt{l}$) is inserted into the schedule to reflect that $\mathtt{l}$ is no longer used as a monitor by $\mathtt{t}$. The global store recorded in the schedule at acquire and release points is used to define safety conditions for mutual exclusion and transactional execution as we describe below.

The semantics makes no attempt to enforce a particular concurrency model on thread execution. Instead, we specify safety properties that dictate the legality of an interleaving by defining predicates on schedules. To do so, it is convenient to reason in terms of *regions*, *i.e.*, subschedules produced as a result of `guard` expression evaluation:

$$\mathit{region}(S) = \{S_i \preceq S \mid S_i = \mathbf{acq}_{\mathtt{t}}^{\Gamma}\,\mathtt{l}.S_i'.\mathbf{rel}_{\mathtt{t}}^{\Gamma'}\,\mathtt{l}\}$$

For any region $R = \mathbf{acq}_{\mathtt{t}}^{\Gamma}\,\mathtt{l}.S_i'.\mathbf{rel}_{\mathtt{t}}^{\Gamma'}\,\mathtt{l}$, $\mathcal{T}(R) = \mathtt{t}$, and $\mathcal{L}(R) = \mathtt{l}$.

For a schedule to be safe with respect to a concurrency control protocol where evaluation of a guarded expression is mediated using a transactional monitor, it must conform to the notions of transactional *atomicity* and *isolation*:

$$\frac{\begin{array}{c}\forall N, R \in \mathit{region}(S), \qquad \mathcal{T}(N) = \mathcal{T}(R) = \mathtt{t} \\ \mathtt{l} = \mathcal{L}(N) \qquad R = S'.N.S'' \\ \mathbf{acq}_{\mathtt{t}'}^{\Gamma}\,\mathtt{l} \notin S'', \qquad \mathtt{t} \neq \mathtt{t}'\end{array}}{\mathit{atomic}(S)} \qquad \frac{\begin{array}{c}\forall R \in \mathit{region}(S) = \mathbf{acq}_{\mathtt{t}}^{\Gamma}\,\mathtt{l}.S'.\mathbf{rel}_{\mathtt{t}}^{\Gamma'}\,\mathtt{l} \\ \forall \mathbf{rd}_{\mathtt{t}}\,\mathtt{l}' \in S' \\ \Gamma(\mathtt{l}) = \sigma \qquad \Gamma'(\mathtt{l}) = \sigma' \\ \sigma(\mathtt{l}') = \sigma'(\mathtt{l}')\end{array}}{\mathit{isolation}(S)}$$

Atomicity guarantees, that the effects of a guarded expression's evaluation are not propagated to other threads until the the end of the region containing operations of this expression. Observe, that our semantics propagates updates to the global store at the end of the region (which is a release operation). These updates become visible to any other thread that subsequently executes a `guard` expression using the same monitor. Thus,

effects of evaluating a guarded expression within an inner region might become visible to other threads before the end of the enclosing region. This would however violate our intuitive notion of atomicity for the enclosing guarded region, since partial effects (*i.e.*, resulting from evaluation performed within the inner region) would become visible before it completes. Our atomicity rule thus captures the essence of the closed nested transaction model: the effects of an inner transaction are visible to the parent, via the local store, but are propagated to other threads only when the outermost transaction completes.

Isolation guarantees that evaluation of the guarded expression is not affected by operations of other (concurrently executing) threads, *i.e.*, values observed during evaluation of the guarded expression do not reflect updates performed by other threads. This property is enforced by requiring that for every location read during evaluation of the guarded expression in region R, the global store associated with the monitor guarding this expression is not modified in R. We define $\mathit{tsafe}(S)$ (read "transaction-safe") to hold if $\mathit{atomic}(S)$ and $\mathit{isolation}(S)$ hold.

The predicate $\mathit{msafe}(S)$ (read "mutual-exclusion-safe") specified below defines the structure of schedules that correspond to an interpretation of monitors in terms of mutual exclusion:

$$\frac{\begin{array}{c}\forall R \in \mathit{region}(S), \mathcal{T}(R) = \mathtt{t}, \mathcal{L}(R) = \mathtt{l} \\ \mathbf{acq}^{\Gamma'}_{\mathtt{t}'}\,\mathtt{l} \notin R, \qquad \mathtt{t} \neq \mathtt{t}'\end{array}}{\mathit{msafe}(S)}$$

For a schedule to be safe with respect to a concurrency control protocol using exclusive monitors, multiple threads cannot be allowed to concurrently evaluate an expression guarded by the same monitor. Thus if thread `t` is evaluating expression `e` guarded by monitor `l`, no other thread can attempt to acquire `l` until `t` releases it.

Mutual-exclusion-safety by itself does not guarantee transaction-safety in the presence of nesting. One can easily construct an *msafe* schedule where neither *atomic* nor *isolation* hold. Fortunately, programmers almost always use mutual exclusion locks to enforce properties of atomicity and isolation. We show that in the case of programs where

mutual exclusion is guaranteed to enforce these properties, both implementations of monitors (transactional and exclusive) can safely and transparently co-exist. Comprehensive empirical evidence proving that this is indeed the case for most Java programs is provided in [20]. The notion of atomicity used in [20] is in fact more restrictive than our notion of safety, which would allow an even larger number of programs to be accepted by our semantics. In our case, the notion of safety is refined by taking into account actual data access operations instead of just the monitor operations.

Suppose program P induces schedule $S_{\mathtt{P}}$ and *tsafe*$(S_{\mathtt{P}})$ holds. Now, if *msafe*$(S_{\mathtt{P}})$ also holds, then any region in $S_{\mathtt{P}}$ could be implemented either transactionally or using mutual exclusion. Suppose, however, that *msafe*$(S_{\mathtt{P}})$ does not hold. This is clearly possible: consider an interleaving in which distinct threads concurrently evaluate guarded expressions protected by the same monitor, but the bodies of the expressions access disjoint locations.

Our soundness theorem shows that every such schedule can be permuted to one which satisfies both *msafe* and *tsafe*. In other words, for every transaction-safe schedule, there is an equivalent schedule that also satisfies constraints defining mutual exclusion. Thus, as long as regions in a program obey atomicity and isolation, they can be implemented either using exclusive or transactional monitors without violating program semantics.

Theorem 6.1.1 *(Soundness)*
Let

$$\mathtt{E}^{\mathtt{t}}_{P}[\mathtt{e}], \Delta_0, \Gamma_0, \phi \Longrightarrow \mathtt{E}^{\mathtt{t}}_{P}[\mathtt{l}], \Delta, \Gamma, S$$

and suppose tsafe(S) *holds but msafe*(S) *does not. Then, there exists a schedule* S_M *such that*

$$\mathtt{E}^{\mathtt{t}}_{P}[\mathtt{e}], \Delta_0, \Gamma_0, \phi \Longrightarrow \mathtt{E}^{\mathtt{t}}_{P}[\mathtt{l}], \Delta_M, \Gamma_M, S_M$$

where tsafe(S_M) *and msafe*(S_M) *hold, and in which* $\Gamma = \Gamma_M$.

Proof Let $R \preceq S = \mathbf{acq}_{\mathtt{t}}^{\Gamma}\,\mathtt{l}.S'.\mathbf{rel}_{\mathtt{t}}^{\Gamma'}\,\mathtt{l}$ and suppose *msafe*(S) does not hold. Since *msafe*(S) does not hold, there must be some $R' = \mathbf{acq}_{\mathtt{t}'}^{\Gamma''}\,\mathtt{l}.S''.\mathbf{rel}_{\mathtt{t}'}^{\Gamma'''}\,\mathtt{l}$ such that $\mathbf{acq}_{\mathtt{t}'}^{\Gamma''}\,\mathtt{l} \in S'$. Since *isolation*$(S)$ holds, none of the effects performed by $\mathtt{t}'$ while synchronized on monitor $\mathtt{l}$ are visible to $\mathtt{t}$. Similarly, since atomicity holds, actions performed by $\mathtt{t}$ within R are not visible to $\mathtt{t}'$ in S''. Suppose that $\mathbf{rel}_{\mathtt{t}'}^{\Gamma'''}\,\mathtt{l}$ follows R in S. Then, effects of R may become visible to operations in R'. But, then *isolation*(R') would not hold. Thus, we can construct a permuted schedule S_M of S in which R' precedes R, *msafe*(S_M), *isolation*(S_M), and *atomic*(S_M) all hold.

6.2 Design

Our design applies the semantics for `guard` expressions to Java monitors to allow co-existence of both transactional and exclusive monitors within the same system. It should be allowed for different synchronized blocks to be protected by either a transactional monitor or an exclusive monitor, without changing execution semantics of a program. The same arguments applies to synchronized methods. The modifications necessary to support this hybrid approach should not lead to performance degradation in the common case of single-threaded execution of a synchronized block (*i.e.*, when monitor is uncontended), and should lead to notable performance improvements in the case when multiple threads attempt to acquire the same monitor simultaneously (*i.e.*, when monitor is contended). Also, the runtime system should be equipped with a mechanism to determine whether to execute a given monitor transactionally or exclusively: mutual exclusion should be used when a monitor is uncontended, transactions should be used only when contention is detected.

Synchronization techniques using exclusive monitors have been thoroughly investigated. Furthermore, recent solutions [3, 5] are optimized towards the non-contended case, which precisely fulfills our design requirement. Bringing transactions to the programming language context is somewhat more challenging and doing it efficiently is still an open issue. Our solution uses optimistic transactions to mediate operations of different threads

acquiring the same transactional monitor. Alternative designs, along with motivation behind the choice of optimistic transactions have been discussed in Section 1.2.3.

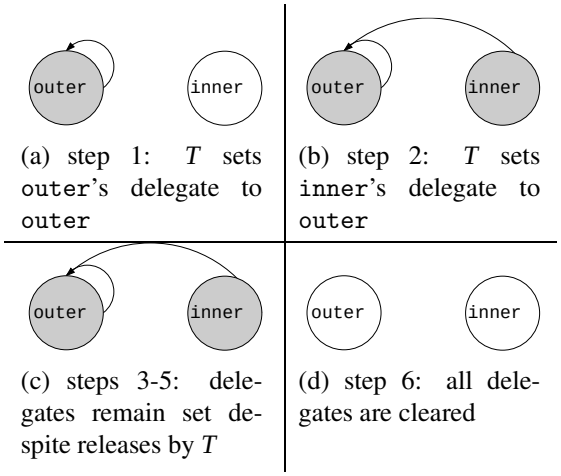
Our discussion makes an obvious but important assumption that while any given monitor can be at one time executed transactionally and at another time exclusively, multiple threads cannot simultaneously acquire it using different protocols. A thread that attempts to acquire a monitor of one type (say, exclusive) currently held by another thread (or threads) in a different mode (say, transactional) will be blocked until the monitor is released by all its holders.

6.2.1 Nesting and Delegation

Since Java monitors support nesting, transactional monitors must also support nesting. There is no conceptual difficulty in dealing with nesting. We have already observed that for a large number of Java programs, the behavior of the synchronization protocol based on closed nested transactions is equivalent to that of the protocol based on mutual exclusion. However, providing support for nesting may pose efficiency challenges since each nested transaction must maintain enough information to guarantee atomicity and isolation of transactional execution.

Note that there is no a priori reason why concurrent data accesses must be mediated exclusively by the monitor protecting them. For example, a single global monitor could conceivably be used to protect all synchronized blocks in the program. Under transactional execution, a single global monitor effectively serves to implement the `atomic` construct [27], described in Chapter 3. Under exclusive execution, a single global monitor defines a global exclusive lock. The primary reason why applications choose not to mediate access to shared data using a single monitor is because of increased contention on monitor acquisition and potentially reduced concurrency. In the case of mutual exclusion, a global lock reduces opportunities for concurrent execution. In the case of transactional execution, a global monitor would have to potentially mediate accesses from logically disjoint transactions, and is likely to be inefficient and non-scalable.

	T	T'
1	**acq**(*outer*)	
2	**acq**(*inner*)	
3		**acq**(*inner*)
4	**rel**(*inner*)	
5	**rel**(*outer*)	
6		**rel**(*inner*)



(a) step 1: T sets `outer`'s delegate to `outer`

(b) step 2: T sets `inner`'s delegate to `outer`

(c) steps 3-5: delegates remain set despite releases by T

(d) step 6: all delegates are cleared

Figure 6.4. Delegation example

Nonetheless, we can leverage this observation to optimize an important specific case for transactional monitors. Consider a thread T acquiring monitor `outer` and, prior to releasing `outer`, acquiring monitor `inner`. If no other thread simultaneously attempts to acquire monitor `inner` (monitor is uncontended), acquisition of monitor `inner` can be *delegated* to monitor `outer`. In other words, instead of synchronizing on monitor `inner` we can establish `outer` as `inner`'s *delegate* and synchronize on `outer`. Since monitor `inner` is uncontended, there is nothing for `inner` to mediate, and no loss of efficiency accrues because of nesting (provided that the act of setting a delegate is inexpensive). Of course, when monitor `inner` is contended, we must ensure that atomicity and isolation are appropriately enforced. Note that if `inner` was an exclusive monitor, there would be no benefit in using delegation since acquisition of an uncontended mutual-exclusion monitor is already expected to have low overhead.

Figure 6.4 is an illustration of how the delegation protocol works for a specific schedule; for simplicity, we show only synchronization operations. The schedule consists of steps 1 through 6 enumerated in the first column of the table describing the schedule. The right-hand side of Figure 6.4 describes the state of the (transactional) monitors, used throughout the schedule, with respect to delegation. A monitor whose delegate has been set is shaded grey, an arrow represents a reference to monitor's delegate. We assume that

delegates of both monitor `outer` and monitor `inner` are initially unset. Thread T starts by entering synchronized block protected by monitor `outer`, creating a new transaction whose accesses are going to be mediated by `outer` and setting a delegate of `outer` to itself (step 1 in Figure 6.4(a)). Then thread T proceeds to enter an inner synchronized block protected by monitor `inner`. Because no delegate for `inner` exists and thread T is already executing within a transaction, T sets a delegate of `inner` to point to monitor `outer` (step 2 in Figure 6.4(b)). Note that the protocol implements a closed nested transaction model: effects of T's execution within the synchronized block guarded by monitor `inner` are not made globally visible until the execution of the outer synchronized block is complete (and its transaction commits), since it is only monitor `outer` that is responsible for mediating concurrent accesses. The delegates stay set throughout steps 3, 4 and 5 (Figure 6.4(b)), even after thread T, the setter of both delegates, releases monitor `outer`. In the meantime however thread T' attempts to enter a different synchronized block guarded by monitor `inner`. The delegate of `inner` is at this point already set to `outer` and thread T' starts its own transaction whose accesses are going to be mediated by monitor `outer`. The delegates are cleared only after transaction executed by thread T' gets terminated (either committed or aborted) – there is no more use for the delegates at this point. Note that some precision is lost in this case: transactional meta-data maintained by `outer` is presumably greater than what would be necessary to simply implement consistency checks for actions guarded by `inner`. However, despite nesting, only one monitor has been used to mediate concurrent data accesses and only one set of transactional meta-data had to be created (associated with monitor `outer`). Note, however, that if actions taken in steps 2 and 3 were performed in the opposite order (*i.e.*, T' would try to acquire monitor `inner` before T), thread T' would try to acquire monitor `inner` whose delegate is not set at the point of acquisition. In this case the protocol would behave in the same way as a traditional implementation of the closed nested transaction model – two sets of transactional meta-data would be created (one for monitor `outer` and one for monitor `inner`) to mediate concurrent data accesses.

6.2.2 Transactions to Mutual Exclusion Transition

A system using optimistic transactions must include support for the revocation mechanism. Revocation procedure typically relies on the ability to undo effects of all transactional computation. In realistic scenarios, however, some actions are non-revocable (*e.g.*, I/O) - their effects cannot be undone. It is also difficult to predict at the point of starting a new transaction if any non-revocable action will become part of this transaction since it is non-trivial, in general, to predict the exact shape of the call graph. One solution is to abort and revoke a transaction before a non-revocable action is about to take place but, in the case the same execution path is chosen after abort, this could lead to repeated (potentially infinitely) aborts. Our system utilizes a different solution. Because we enable coexistence of both transactional and exclusive monitors, transactional execution can attempt a transition to mutual-exclusion right before executing a non-revocable action. This way a costly revocation of transactional monitors can be avoided and computation may safely proceed. An identical solution is used to thread notifications – mode transition is attempted before executing `wait` and `notify` actions.

In order to enable mode transition, every thread executing a synchronized block protected by a transactional monitor must maintain an ordered (in the order of acquisition) list of all inner transactional monitors. At the transition point a thread must atomically leave and successfully commit all the transactions it is currently executing as well as successfully acquire all monitors from the list in mutual exclusion (turning them into mutual exclusion monitors). Successful acquisition of mutual-exclusion monitors involves waiting on all threads holding these monitors to release them (and cleanup the respective delegates). From the point of a successful transition, a thread keeps executing in mutual-exclusion mode until exiting the outermost synchronization block. If the transition is unsuccessful, the transactional monitor (along with all the inner monitors) must be revoked and re-executed.

6.3 Implementation

An efficient implementation of exclusive monitors (*i.e.*, thin-locks [5]) already exists in Jikes RVM. Therefore we concentrate our description on the implementation of transactional monitors. We use transactional delegation protocol to reduce overheads for uncontended inner transactional monitors by deploying nested transaction support only when absolutely necessary. Thus, transactions are leveraged only at the outermost level or in the case of contended inner monitors.

Our implementation follows the three-phase optimistic approach for closed nested transactions. We use transaction-local, per-object versions to log shared data accesses. In the write phase (at commit time), the updates are propagated to the shared heap lazily, using forwarding pointers as described in Section 2.2. In the closed nested transaction model the effects of inner transactions are not visible until the outermost transaction commits. In other words, termination of an inner transaction can be deferred until the outermost transaction terminates. At this point, the validation phase takes place and all transactions are examined to decide if they should commit or abort. Adopting this approach enables us to maintain versions only per outermost transaction scope. In the remainder of this section we discuss our strategy for detection of serializability violations (through data dependency tracking), our solutions for the automatic revocation procedure and shared data versioning, as well as some additional implementation details. Support for shared data management is provided using read and write barriers (as described in Section 2.4) inserted by Jikes RVM's compilers.

6.3.1 Dependency Tracking

The semantics of Java monitors dictates that data dependencies are tracked only between transactions whose operations are mediated by the *same* transactional monitor. Support for dependency tracking is based on the notion of access maps described in Section 2.3. Each transaction hashes its shared data accesses into two local maps: a read map and a write map. Once a transaction commits and propagates its updates into the shared

heap it also propagates information about its own updates to a global write map associated with the monitor. Other transactions whose operations are mediated by the same monitor will then, during their validation phase, intersect their local read maps with the global write map to determine if the shared data accesses caused a violation of serializability. Note however that monitors (and thus transactions) can be nested. Since all transactions are terminated (and thus validated) at the same time, the local maps can be maintained only per outermost transaction scope. In other words both local maps become associated with a thread when starting the outermost transaction and the association is cleared when the outermost transaction terminates. However, because of nesting, there may exist multiple global write maps associated with inner monitors. The validation phase must then check the local read map against all the global write maps. In order to reduce the number of read barriers we apply the optimization described in Section 5.3.1 to execute read barriers only on reference loads: objects referenced from the thread's stack are pre-read before the acquisition of a transactional monitor. This read barrier optimization enables early detection of serializability violations as a direct result of conservatively pre-reading objects that are never read, but only written to (the details are described below).

6.3.2 Revocation

Our revocation procedure is identical to the one used for revocable monitors, as described in Section 4.2.2, and allows for a transaction to be aborted at an arbitrary point during its execution. The abort is signaled by throwing the `Revoke` exception. Undo and re-execution procedures are implemented using a combination of bytecode re-writing and virtual machine modifications. Even though Java monitors are lexically scoped, it is necessary to support transaction aborts at arbitrary points to correctly handle native method calls as well as `wait` and `notify` operations, as described in Section 6.2.2.

6.3.3 Versioning

We use shared data versioning to prevent the effects of incomplete transactions from being made visible to other threads until they commit. The implementation of versioning is based on a general procedure described in Section 2.2.2 using lazy propagation of updates.

Because our array versioning procedure is identical to that used for versioning objects, we refer only to objects in the following description. Versions are accessible through a forwarding pointer from the original object. We use a "copy-on-write" strategy for creating new versions. A transaction creates a new (*uncommitted*) version right before performing its first update and redirects all subsequent read and write operations to access that version. It is important to remember that in order to guarantee transparency all programs executed in our system must satisfy certain safety properties (defined in Section 6.1), similar to the notion of atomicity as defined in [21] and [20]. In most cases atomicity also implies race-freedom, that is, every access to a shared data item being protected by the same monitor. As a result, two writes to the same location performed by two different threads are very likely to be protected by the same transactional monitor – they are automatically detected as a serializability violation. Because of pre-reading optimization mentioned above the writes are also treated as reads which would result in the same slot of both local read and write maps being tagged, and subsequent abort of one transaction upon successful commit of the other. Therefore, only the first transaction writing to a given object needs to create a version for it. Other transactions are immediately aborted.

Upon successful commit of a transaction, the current version becomes the *committed version* and remains accessible via the forwarding pointer from the original object. All subsequent accesses are re-directed via the forwarding pointer to the committed version. When a transaction aborts all its versions are discarded. Note that at most two versions of an object exist at any given time: a committed version and an uncommitted version.

As noted above, the read barriers are only executed on reference loads. In general, multiple on-stack references may end up pointing to different versions of the same object. This is possible, even though read barriers are responsible for retrieving the most

up-to-date version of the object, because updates can be performed concurrently. It is the responsibility of the run-time system to ensure that the version of an object accessible through an on-stack reference is the up-to-date. The visibility rules for the Java Memory Model [43] (described in Section 4.1.2) dictate that at certain synchronization points (*e.g.*, monitor entry, access to volatile variables, *etc.*) threads are obliged to have the same view of the shared heap. As a result, it is legal for operations mediated by a transactional monitor to access out-dated versions of objects until such a synchronization point is reached. It is only at these points that all the references residing on the stack need to be forwarded to refer to the most up-to-date versions of their respective objects. Reference forwarding is implemented using a stack inspection procedure described in Section 5.3.3.

In addition to performing reference forwarding at synchronization points, when a version is first created by a transaction, the thread creating this version must forward all references on its stack to point to the new version. This ensures that all subsequent accesses (by the same thread) observe the results of the update. Reference forwarding is also used when a transaction aborts to remove all the newly created versions from the stack.

We now present an example of how these different implementation features interact. Figure 6.6 describes actions concerning shared data versioning and serializability violation

	T	T'	T''
1	**acq**(`outer`)		
2	**wt**(`o2`)		
3			**acq**(`inner`)
4			**wt**(`o1`)
5		**acq**(`outer`)	
6		**acq**(`inner`)	
7		**rd**(`o1`)	
8	**rel**(`outer`)		
9			**rel**(`inner`)
10		**rd**(`o1`)	
11		**rel**(`inner`)	
12		**rel**(`outer`)	

Figure 6.5. A non-serializable schedule.

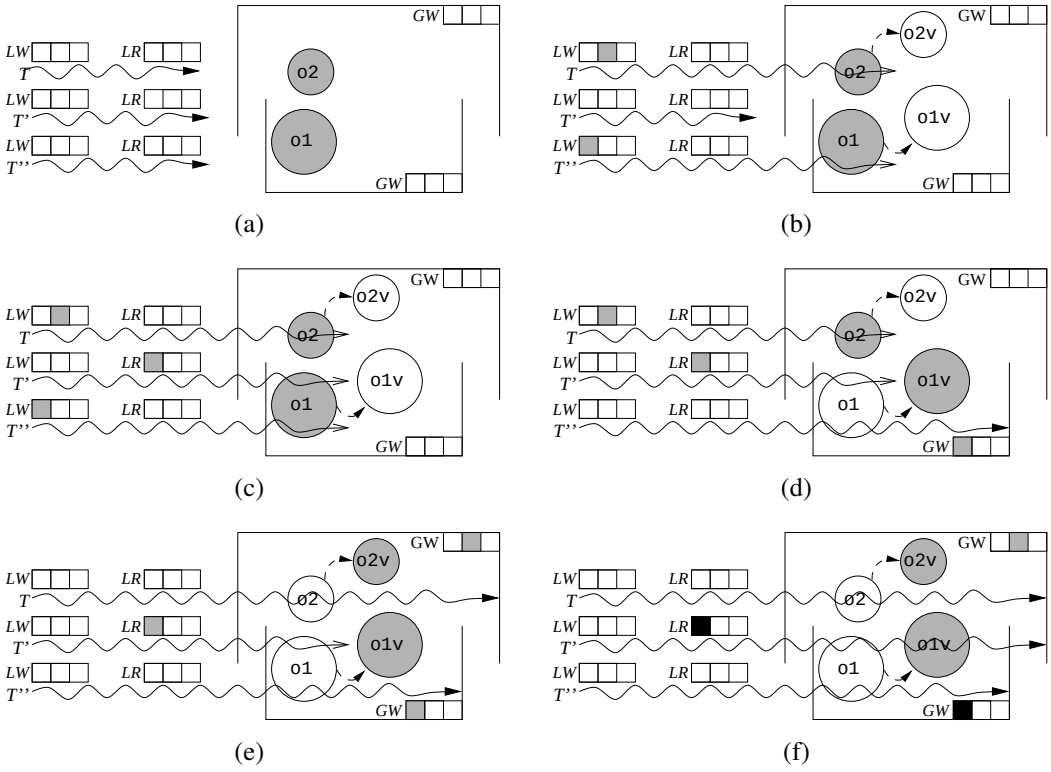


Figure 6.6. A non-serializable execution.

detection, performed by threads T, T' and T'' executing the schedule shown in Figure 6.5. The diagram in Figure 6.6(a) represents the initial state, before any threads have started executing. Wavy lines represent threads, and circles represent objects `o1` and `o2`. The objects have not yet been versioned – they are shaded grey because at the moment they contain the most up-to-date values. The larger box (open at the bottom) represents the scope of transactional monitor `outer`, the smaller box (open at the top) represents the scope of transactional monitor `inner`. Both the global write map (*GW*) associated with the monitor and the local maps (write map *LW* and read map *LR*) associated with each thread have three slots. Maps that belong to a given thread are located above the wavy line representing this thread. We assume that accesses to object `o1` hash to the first slot of every map and accesses to object `o2` hash to the second slot of every map

The execution begins with threads T and T'' starting to run transactions whose operations are mediated using monitors `outer` and `inner` (respectively) and performing updates

to objects o2 and o1 (respectively), as presented on Figure 6.6(b). Updating objects involves version creation (o2v becomes a new version of object o2 and o1v becomes a new version of object o1) and tagging of the local write maps. Thread T tags the second slot of its local write map since it modifies object o2, whereas thread T'' tags the first slot of its local write map since it modifies object o1. On Figure 6.6(c) thread T' starts executing: it starts running the outermost transaction whose operations are mediated by monitor outer, its inner transaction whose operations are mediated by monitor inner and performs a read of object o1 (reading an object involves tagging thread's local read map). In the next diagram (Figure 6.6(d)), transaction executed by thread T'' attempts a commit. Since no writes by other transactions mediated by monitor inner have been performed, commit is successful: o1v becomes the committed version, contents of the local write map associated with T'' is transfered to inner's global write map and the local write map is cleared. Similarly, on Figure 6.6(e), transaction executed by thread T commits successfully: o2v becomes a committed version and the local write map is cleared after its contents has been transfered to the global write map associated with monitor outer. On Figure 6.6(f) thread T' proceeds to again read object o1 and then to commit its transactions (both inner and outer). However, because a new committed version of object o1 has been created, o1v is read by T' instead of the original object. When attempting to commit both its inner and the outer transactions, thread T' must intersect its local read map with global maps associated with both monitor outer and monitor inner. The first intersection is empty (no writes performed in the scope of monitor outer could compromise reads performed by T'), the second however is not – both transactions executed by T' must be then aborted and re-executed.

6.3.4 Header Compression

For performance we need efficient access to several items of meta-data associated with each object (*e.g.*, versions and their identities, delegates, access maps associated with objects representing monitors, *etc.*). At the same time, we must keep overheads to a

minimum when transactions are not used. The simplest solution is to extend object headers to associate the necessary meta-data. Our transactional meta-data requires up to four 32-bit words. Unfortunately, our virtual machine platform does not support variable header sizes and extending the header of each object by four words has serious implications for space and performance, even in the case of non-transactional execution. On the other hand keeping meta-data "on the side" (*e.g.*, in a hash table), also results in a significant performance hit.

We therefore implement a compromise. The header of every object is extended by a single *descriptor word* that is lazily populated when meta-data needs to be associated with the object. If an object is never accessed in a transactional context, its descriptor word remains empty. Because writes are much less common than reads, we treat the information needed for reads as the common case. The first transactional read will place the object's identity hash-code in the descriptor (the run-time generates its own hash-codes to improve data distribution in access maps). If additional meta-data needs to be associated with the object (*e.g.*, a new version on write) then the descriptor word is overwritten with a reference to a new *descriptor object* containing all the necessary meta-data (including the hash-code originally stored in the descriptor word). We discriminate these two cases using the low-order bit of the descriptor word.

6.3.5 Code Duplication

Transactional support (*e.g.*, read and write barriers) is required only when a thread decides to execute a given monitor transactionally. However, it is difficult to determine if a particular method is going to be used in a non-transactional context only. To avoid unnecessary overheads during non-transactional execution, we use bytecode rewriting to duplicate the code of all (user-level) methods actually being executed by the program. Every method can then be compiled in two versions: one that embeds transactional support (transactional methods) and one that does not (non-transactional methods). This allows the run-time system to dynamically build a call chain consisting entirely of non-transactional

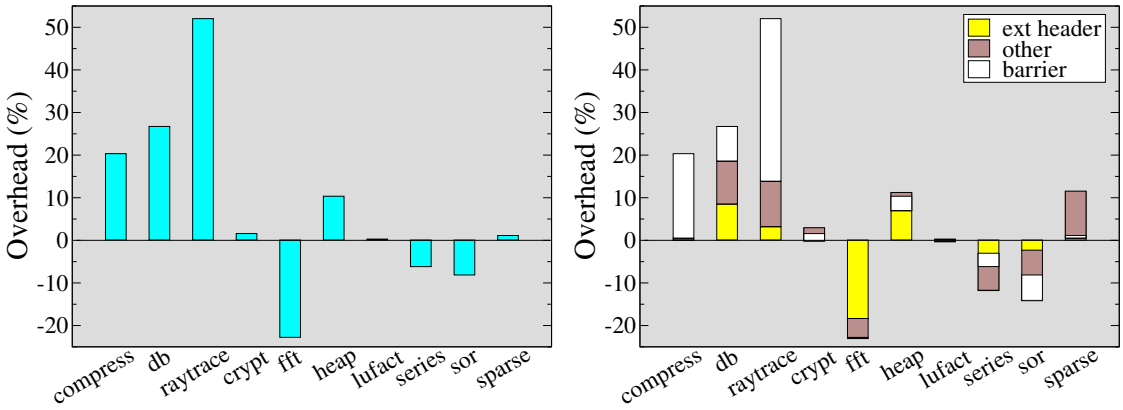
methods for non-transactional execution. Unfortunately, because of our choice to access most up-to-date versions of objects through forwarding pointers, we cannot fully eliminate read barriers even in non-transactional methods. We can however eliminate all write barriers and make the non-transactional read barriers very fast in the common case – they need simply differentiate objects that have never been accessed transactionally from those that have. In addition to the usual reference load, such barriers consist only of a null check, one condition, and one load. These instructions verify that the descriptor word is empty, indicating that the object has never been accessed transactionally, and thus no alternate version has ever been created.

6.3.6 Triggering Transactional Execution

Our implementation must be able to determine whether to execute a given monitor transactionally or exclusively. We use a very light-weight heuristic to detect monitor contention and trigger transactional execution only for contended monitors . The first thread to enter a monitor always executes the monitor exclusively. It is only after a thin mutual-exclusion lock is "inflated" by being turned into a fat lock (on contended acquisition of the lock) that the monitor in question is asserted to be contended. All threads queued waiting for the monitor will then execute transactionally once the currently executing (locking) thread exits the monitor. We recognize that there are more advanced and potentially more conservative heuristics that a production system may wish to use. For example, programmer annotations could be provided to mark the concurrency control mechanism that is to be used for different monitors. Adaptive solutions based on dynamic profiling or solutions utilizing off-line profiles may also provide more refined information on when to best execute monitors transactionally.

6.4 Experimental Evaluation

The performance evaluation of our prototype implementation is divided into two parts. We use a number of single-threaded benchmarks (from the SPECjvm98 [55] and Java



(a) total overhead (b) component overheads

Figure 6.7. Uncontended execution

Grande [54] suites) to measure the overheads of supporting hybrid-mode execution (*e.g.*, compiler-inserted barriers, code-duplication, object layout modifications, *etc.*) when monitors are uncontended. We also use an extended version of the OO7 object database benchmark [13], to expose the range of performance when executing under different levels of monitor contention. We measure the behavior when all monitors are executed transactionally and when using a hybrid scheme that executes monitors transactionally only when sufficient monitor contention is detected. Our measurements were taken on an 700MHz Intel Pentium III symmetric multi-processor (SMP) with 2GB of RAM running Linux kernel version 2.4.20-31.9smp (RedHat 9.0). Our implementation uses version 2.3.4+CVS (with 2005/12/08 15:01:10 UTC timestamp) of Jikes RVM for all the configurations used to take the measurements (mutual-exclusion-only, transactions-only and hybrid). We ran each benchmark configuration in its own invocation of the virtual machine, repeating the benchmark six times in each invocation, and discarding the results of the first iteration, in which the benchmark classes are loaded and compiled, to eliminate the overheads of compilation.

6.4.1 Uncontended Execution

A summary of our performance evaluation results when monitors are uncontended is presented in Figure 6.7. Our current prototype implementation is restricted to running bytecodes compiled with debugging information for local variables; this information is needed by the bytecode rewriter for generating code to store and restore local state in case of abort. Therefore, we can only obtain results for those SPECjvm98 benchmarks for which source code is available.

In Figure 6.7(a) we report total summary overheads for a configuration that supports hybrid-mode execution. The overheads are reported as a percentage with respect to a "clean" build of the "vanilla" unmodified Jikes RVM. The average overhead is on the order of 7%, with a large portion of the performance degradation attributed to execution of the compiler-inserted barriers, as described below. Figure 6.7(b) reveals how different mechanisms for transactional execution affect performance in the uncontended case. The bottom of every bar represents the effect of extending the header of every object by one word (as needed to support transaction-related meta-data). The middle of every bar represents the cost of all other system modifications, excluding compiler-inserted barriers.[2] The top bar captures overhead from execution of the barriers themselves (mostly read barriers but also barriers on static variable accesses).

Observe that changing the object layout can by itself have a significant impact on performance. In most cases, the version of the system with larger object headers indeed induces overheads over the clean build of Jikes RVM, but in some situations (*e.g.*, `FFT` or `Series`), its performance actually improves over the clean build by a significant amount; variations in cache footprint is the most likely cause. The performance impact of the compiler-inserted barriers is also clearly noticeable, especially in the case of benchmarks from the SPECjvm98 suite. When discounting overheads related to the execution of the barriers, the average overhead with respect to the clean build of Jikes RVM drops to a little over 1% on average. This result is consistent with that reported by Blackburn and

[2]The measurements were taken after artificially removing compiler-inserted barriers from the "full" version of the system. Naturally our system cannot function without barriers.

Table 6.1
Component organization of the OO7 benchmark

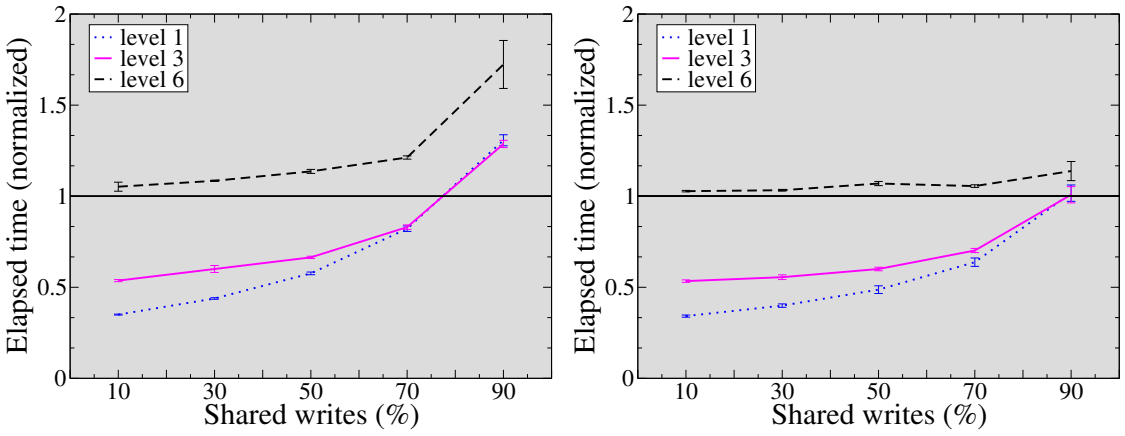
Component	Number
Modules	1
Assembly levels	7
Subassemblies per complex assembly	3
Composite parts per assembly	3
Composite parts per module	500
Atomic parts per composite part	20
Connections per atomic part	3
Document size (bytes)	2000
Manual size (bytes)	100000

Hosking [8] for garbage collection read barriers that can incur overheads up to 20%. It would be beneficial for our system to use a garbage collector that might help to amortize the cost of the read barrier. Fortunately, there exist modern garbage collectors (*e.g.*, [6]) that fulfill this requirement.

6.4.2 Contended Execution

Our motivation behind choosing OO7 as a benchmark was to accurately gauge the various trade-offs inherent with our implementation over a wide range of different workloads, rather than emulating specific workloads of potential applications. We believe the benchmark captures essential features of scalable concurrent programs that can be used to quantify the impact of the design decisions underlying our implementation. The structure of the OO7 database as well as operations performed during database traversals are described in Section 5.4.2.

Our experiments here use traversals that always operate on the *shared* module, since we are interested in the effects of contention on performance of our system. Our implementation of OO7 conforms to the standard OO7 database specification. All the results are obtained with an OO7 database configured as in Table 6.1. Our traversals differ from the original OO7 traversals in adding a parameter that controls entry to monitors at vary-



(a) transactions-only (b) hybrid-mode

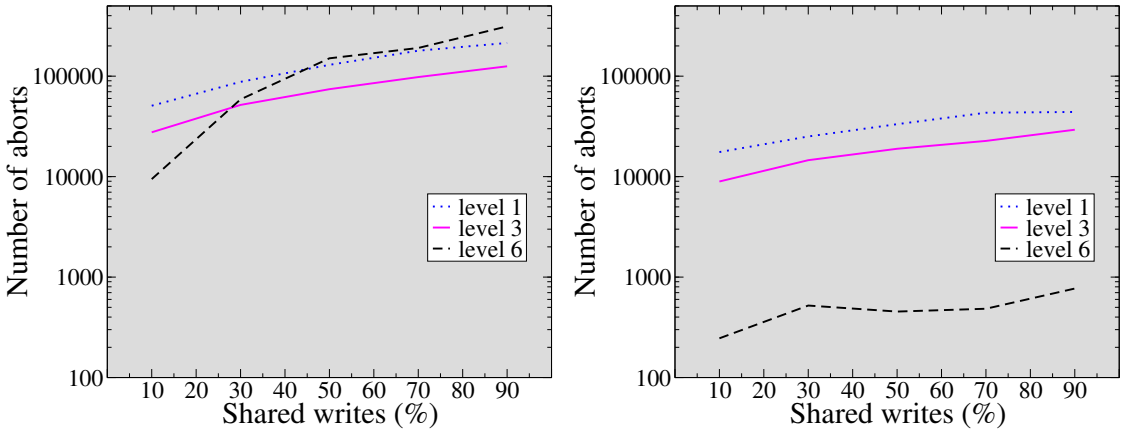
Figure 6.8. Normalized execution times for the OO7 benchmark

ing levels of the database hierarchy. We run 64 threads on 8 physical CPUs. Every thread performs 1000 traversals (enters 1000 monitors) and visits 4M atomic parts during each iteration. When running the benchmarks we varied the following parameters:

- ratio of shared reads to shared writes: from 10% shared reads and 90% shared writes (mostly read-only workload) to 90% shared reads and 10% shared writes (mostly write-only workload).

- level of the assembly hierarchy at which monitors are entered: level one (module level), level three (second layer of complex assemblies) and level six (fifth layer of complex assemblies). Varying the level at which monitors are entered models different granularities of user-level synchronization from coarse-grained through to fine-grained and diversifies the degree of monitor contention.

Figure 6.8 plots execution times for the OO7 benchmark when all threads execute all monitors transactionally (Figure 6.8(a)) and when threads execute in hybrid mode, where the mode is chosen based on monitor contention (Figure 6.8(b)). The execution times are normalized with respect to the performance of the "clean" build of Jikes RVM (90% confidence intervals are also reported). Figure 6.9 plots the total number of aborts for both transactions-only (Figure 6.9(a)) and hybrid (Figure 6.9(b)) executions. Different lines on



(a) transactions-only (b) hybrid-mode

Figure 6.9. Total number of aborts for the OO7 benchmark

the graphs represent different levels of user-level synchronization granularity – one being the most coarse-grained and six being the most fine-grained.

When there is a suitable amount of monitor contention, and when the number of writes is moderate, transactional execution significantly outperforms mutual exclusion by up to three times. The performance of the transactions-only scheme degrades as the number of writes increases (and so does the number of generated hash-codes) since the number of bitmap collisions increases, leading to a large number of aborts even at low contention (Figure 6.9(b)). Extending the size of the maps used to detect serializability violations would certainly remedy the problem, at least in part. However, we cannot use maps of an arbitrary size. This could unfavorably affect memory overheads (especially compared to mutual-exclusion locks) but more importantly we have determined that the time to process potentially multiple maps at the end of the outermost transaction must be bounded. Otherwise, the time spent to process them becomes a source of significant delay (currently each map contains over 16,000 slots). The increased number of aborts certainly has a very significant impact on the difference in performance between the transactions-only and hybrid schemes. The overheads of the transactions-only scheme cannot however be attributed only to the increased abort rate – observe that the shape of the graphs plotting execution times and aborts are different. During hybrid-mode execution, monitors are exe-

cuted transactionally only when monitor contention is detected, read and write operations executed within uncontended monitors incur little overhead, and the revocations are very few. Thus, instead of performance degradation of over 70% in the transactions-only case when writes are dominant, our hybrid scheme incurs overhead on the order of only 10%.

6.5 Related Work

Our system supporting hybrid execution relies on the existence of a low-cost implementation of mutual-exclusion monitors to handle cases when monitors are uncontended (at most one thread attempts to acquire a given monitor at a time). One of the solutions fulfilling our requirements (and in fact used as part of our system), *thin locks*, has been proposed by Bacon *et al.* [5]. Their design goal was to enable low-cost acquisition of uncontended monitors without compromising performance in the case when monitors are contended (multiple threads compete for acquisition of the same monitor). Their solution is optimized towards the common case, that is, the acquisition of an uncontended monitor with no subsequent signaling operations (`wait` or `notify`) executed with respect to this monitor.

In the common case, all the information concerning the monitor state is stored in a header of an object used as a monitor. This includes the recursion count (the same monitor may be acquired multiple times by the same thread) and the identifier of the thread that acquired this monitor. Modifications to the object header are performed using the atomic compare-and-swap operation (its availability in the implementation platform is assumed). This results in the cost of monitor acquisition in the common case to be limited to only several assembly instruction. If more than one thread attempts to acquire the same monitor, or a signaling operation is to be invoked with respect to this monitor, the monitor gets *inflated*. A data structure representing the inflated monitor is created and a header of an object used as a monitor is made to point to this structure. In addition to the same information as the non-inflated monitor, the inflated one maintains two queues for threads waiting on this monitor – one for threads waiting to be notified and one for threads competing to

acquire it. Bacon *et al.* implement their solution in IBM's production Java virtual machine for AIX and in Jikes RVM.

Another approach to implementing Java monitors, *meta-locks*, sharing similar goals with the solution presented above, has been described by Agesen *et al.* [3]. Their solution, similarly to thin locks, is optimized towards the uncontended case. A synchronization operation on a given monitor is in their system divided into three parts: acquisition of the meta-lock associated with the monitor, manipulation of synchronization data associated with the monitor and release of the meta-lock.

In order to minimize the per-object space overhead (even further than in the case of thin locks which required several bits of information even in the common case), they encode monitor state using two lowest-order bits of the *multi-use* word in the header of an object used as a monitor. In case object is never used as a monitor, the multi-use word contains information unrelated to monitor acquisition (*e.g.*, object's hash-code). Otherwise, in the common uncontended case, the content of the multi-use word is atomically replaced with a reference to a fresh *lock record*, containing all monitor-related information (such as identity of a thread acquiring the monitor, recursion count, *etc.*). This operation is implemented using an atomic compare-and-swap and conceptually merges the first two phases of the meta-lock protocol (described previously) into one. In the case of multiple threads competing for acquisition of the same monitor, making the multi-use word to point to the lock record involves several intermediate steps to guarantee correctness of thread interaction in presence of concurrency. The meta-lock protocol has been implemented in the EVM, Sun's production Java virtual machine embedded in Java 2 SDK Production Release.

In order for transactional monitors and exclusive monitors to safely and transparently co-exist in our system, programs executed in the system must satisfy certain safety properties (defined in Section 6.1), similar to the notion of atomicity as defined by Flanagan *et al.* in [21] and [20]. A block of code is considered to be atomic if any possible interaction between operations of this block executed by one thread and operations of other threads is benign. In other words, it appears as if the operations of an atomic block were not inter-

leaved with operations of any other thread. The aforementioned research efforts exploring the notion of atomicity concentrate on verification of whether this property holds for concurrent programs using mutual exclusion as a synchronization mechanism – we provide the details below. We have also described two systems that dynamically enforce atomicity property using transactions as a concurrency control mechanism, one in Chapter 3 and the other in Section 4.4.

Both dynamic and static approaches to atomicity verification are inspired by Lipton's theory of reduction [39]. Let a specific execution of a program be defined as an ordered sequence of interleaved thread operations. Lipton's theory of reduction classifies thread operations into two categories, based on their behavior in all executions of a given program. An operation is a *right mover* if it can be swapped with an operation of another thread immediately following it in the execution order, without changing the behavior of the program (*e.g.*, two reads of two different threads could be safely swapped). An operation of a thread is a *left mover* if it can be swapped, in a similar fashion, with an operation of another thread immediately preceding it in the execution order. A block of code is then considered to be atomic if it consists of a sequence of right movers, followed by a single action, followed by sequence of left movers. By definition, during an arbitrary execution of a program, operations of the atomic block can then be (conceptually) permuted (by swapping them with operations of other threads) to form a contiguous sequence of operations, without changing the behavior of the program. In other words, with respect to the operations of the atomic block, the behavior of the program is the same regardless of their interleaving with operations of other threads.

The static atomicity checker, developed by Flanagan and Quadeer [21], has been built on top of the static type checker. The type system classifies lock acquisition operations as right movers and lock release operations as left movers. An access to a shared variable is considered to be both left and right mover, provided that all accesses to this variable are guaranteed to be protected by the same lock. This property can be automatically verified by the type checker – the association between a shared variable and a lock that should be used to protect accesses to this variable is established using a number of simple

heuristics or manually by a programmer. A programmer is additionally responsible for specifying blocks of code to be checked for atomicity. Atomicity of the code blocks may be then automatically verified by the type checker using classification of operations described above.

The dynamic atomicity checker (the Atomizer), developed by Flanagan and Freund [20], instruments the code of an application to allow for tracking of thread operations at run-time, and thus atomicity verification is performed with respect to specific executions. The approach to classify thread operations is similar to the one used by the static checker, but a programmer is relieved from a burden of explicitly specifying locks protecting shared variables. Locking discipline is automatically inferred by the atomicity checker. Annotations specifying which blocks of code should be checked for atomicity are also optional. By default, all synchronized blocks of code (including synchronized methods) as well as public and protected methods are verified. The results of verifying atomicity for a significant number of Java benchmarks (over 100,000 lines of Java code) confirm the hypothesis of the authors that, for the majority of Java programs analyzed, mutual exclusion is indeed correctly used to enforce the atomicity property.

6.6 Conclusions

Existing approaches to providing concurrency abstractions for programming languages offer disjoint solutions for mediating concurrent accesses to the shared data throughout the lifetime of the entire application. Typically these mechanisms are either based on mutual exclusion or on some form of transactional support. Unfortunately, none of these techniques is ideally suited for all possible workloads. Mutual exclusion performs best when there is no contention on guarded region execution, while transactions have the potential to extract additional concurrency when contention exists.

We have designed and implemented a system that seamlessly integrates mutual exclusion with a synchronization mechanism using optimistic transactions. We also formally argue correctness (with respect to language semantics) of such a system and provide a

detailed performance evaluation of our hybrid scheme for different workloads and varying levels of contention. We demonstrate that our implementation of the system supporting hybrid execution has low overheads (on the order of 7%) in the uncontended (base) case and that significant performance improvements (speedups up to $3\times$) can be expected from running contended monitors transactionally.

7 CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

The goal stated in the thesis statement was to use optimistic transactions for managing concurrency in a programming language environment and demonstrate improvements over concurrency management techniques based on mutual exclusion. We believe that this goal has been achieved. We have presented revocable monitors, described their ability to resolve deadlock and priority inversion problems and demonstrated their effectiveness in improving throughput of high-priority threads in a priority scheduling environment. Safe futures allow concurrent programs to be constructed often through only a small re-write of the sequential code. Our implementation of this mechanism allows concurrency to be extracted for applications with modest mutation rates, which we have confirmed through experimental evaluation. Transactional monitors provide seamless integration between mutual exclusion based synchronization and transactional synchronization. We have presented a low-overhead implementation of transactional monitors and demonstrated that significant performance improvements can be expected in comparison to using traditional exclusive monitors.

7.2 Future Work

One of the main goals for our future work is to better understand implications of the current design and implementation decisions and use this knowledge for even further improved performance. First of all, we would like to contrast our current approach with a solution using pessimistic transactions, preferably implemented within the same platform for easy comparison. The same argument applies to other aspects of our approach, including the choice of logging, barrier-related trade-offs. Ultimately, we would like to construct

a generic framework where all the components are freely interchangeable which would allow for an easy analysis of their mutual interactions and their final effect on the overall performance.

We would also like to explore application of compiler optimizations and hardware support to further improve performance of our solutions. For example, the ability to statically identify thread-local objects would be a great help in reducing barrier-related overheads. Hardware support could facilitate detection of shared data dependencies (*e.g.*, through modifications to cache coherency protocols) or lower the overheads associated with revocations (*e.g.*, through hardware-assisted thread checkpointing).

Another direction we would like to pursue is to experiment with semantics-related trade-offs. Perhaps programmability could be improved by exposing certain mechanisms to a programmer, for example, through parameterizing revocation actions where a programmer is responsible for providing a routine that is invoked before, after or instead of simply re-executing a transaction. Lowering a degree of transparency could also lead to performance improvements, for example, if only pre-specified objects would be subject to transactional access control.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Latte : An open-source Java virtual machine and just-in-time compiler.

[2] The Ovm virtual machine.

[3] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White. An efficient meta-lock for implementing ubiquitous synchronization. In OOPSLA'99 [46], pages 207–222.

[4] Bowen Alpern, C. R. Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Ton Ngo, Mark Mergen, Janice C. Shepherd, and Stephen Smith. Implementing Jalapeño in Java. In OOPSLA'99 [46], pages 314–324.

[5] David Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: Featherweight synchronization for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, volume 33, pages 258–268, May 1998.

[6] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, volume 38, pages 285–298, January 2003.

[7] A.J. Bernstein. Program analysis for parallel processing. *IEEE Transactions on Computers*, 15(5):757–762, October 1966.

[8] Stephen M. Blackburn and Antony L. Hosking. Barriers: Friend or foe? In *Proceedings of the ACM International Symposium on Memory Management*, pages 143–151. ACM, 2004.

[9] Bruno Blanchet. Escape analysis for object-oriented languages: Application to Java. In OOPSLA'99 [46], pages 20–34.

[10] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In OOPSLA'99 [46], pages 35–46.

[11] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. The real-time specification for Java, June 2000.

[12] Michael J. Carey, David J. DeWitt, Chander Kant, and Jeffrey F. Naughton. A status report on the OO7 OODBMS benchmarking effort. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 29, pages 414–426, October 1994.

[13] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. In *Proceedings of the ACM International Conference on Management of Data*, volume 22, pages 12–21, June 1993.

[14] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In OOPSLA'99 [46], pages 1–19.

[15] Laurent Daynès and Grzegorz Czajkowski. Lightweight flexible isolation for language-based extensible systems. In *Proceedings of the International Conference on Very Large Data Bases*, 2002.

[16] Zeng Fancong. Deadlock resolution via exceptions for dependable Java applications. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 731–740, 2003.

[17] Zeng Fancong and Richard P. Martin. Ghost locks: Deadlock prevention for Java. In *Proceedings of the Mid-Atlantic Student Workshop on Programming Languages and Systems*, 2004.

[18] Cormac Flanagan and Matthias Felleisen. The semantics of future and its use in program optimizations. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 209–220, 1995.

[19] Cormac Flanagan and Matthias Felleisen. The semantics of future and an application. *J. Funct. Program.*, 9(1):1–31, 2005.

[20] Cormac Flanagan and Stephen N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 256–267, 2004.

[21] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 338–349, 2003.

[22] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183. ACM Press, 1998.

[23] James Gosling, Bill Joy, Guy Steele, Jr., and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000.

[24] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Data Management Systems. Morgan Kaufmann, 1993.

[25] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[26] Per Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, April 1970.

[27] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 38, pages 388–402, November 2003.

[28] Tim Harris and Keir Fraser. Revocable locks for non-blocking programming. In PPOPP'05 [47], pages 72–82.

[29] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In PPOPP'05 [47], pages 48–60.

[30] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 92–101, 2003.

[31] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[32] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, May 1996. Chapter on distributed collection by Lins.

[33] Simon L. Peyton Jones, Andrew Gordon, and Sigbjòrn Finne. Concurrent Haskell. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 295–308, 1996.

[34] JSR166: Concurrency utilities. http://www.jcp.org/en/jsr/detail?id=166.

[35] M. Katz. Paratran: A transparent, transaction based runtime mechanism for parallel execution of Scheme. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1989.

[36] David Kranz, Robert H. Halstead, Jr., and Eric Mohr. Mul-T: A high-performance parallel Lisp. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, volume 24, pages 81–90, July 1989.

[37] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 9(4):213–226, June 1981.

[38] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.

[39] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, December 1975.

[40] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3), March 1988.

[41] Barbara Liskov and Liuba Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, volume 23, pages 260–267, July 1988.

[42] Jeremy Manson, Jason Baker, Antonio Cunei, Suresh Jagannathan, Marek Prochazka, Bin Xin, and Jan Vitek. Preemptible atomic regions for real-time Java. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 62–71, 2005.

[43] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 378–391, 2005.

[44] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 185–197, 1990.

[45] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, April 1981. Also published as MIT Laboratory for Computer Science Technical Report 260.

[46] *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 34, October 1999.

[47] *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005.

[48] Polyvios Pratikakis, Jaime Spacco, and Michael W. Hicks. Transparent proxies for Java futures. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 39, pages 206–223, October 2004.

[49] Carlos Garcia Quiñones, Carlos Madriles, Jesùs Sànchez, Pedro Marcuello, Antonio Gonzàlez, and Dean M. Tullsen. Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 269–279, 2005.

[50] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. Jade: A high-level, machine-independent language for parallel programming. *IEEE Computer*, 26(6):28–38, 1993.

[51] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Ben Hertzberg. A high performance software transactional memory system for a multi-core runtime. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2006.

[52] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 29(9):1175–1185, September 1990.

[53] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[54] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel Java Grande benchmark suite. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, page 8, 2001.

[55] SPEC. SPECjvm98 benchmarks, 1998. http://www.spec.org/osg/jvm98.

[56] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Transactional monitors for concurrent objects. In Martin Odersky, editor, *Proceedings of the European Conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 519–542. Springer-Verlag, 2004.

VITA

VITA

Adam Welc was born and grew up in Bydgoszcz, a city located in the northern part of Poland, where he also attended high school at Nicolas Copernicus II General Secondary School. He received a Master of Science in Computer Science from the Poznan University of Technology, Poland, in May 1999, and a Master of Science in Computer Science from the Purdue University in May 2003.

In January 2000, Adam became a research assistant in the Secure Software Systems group at the Purdue Department of Computer Sciences, working with Professor Antony Hosking and Professor Suresh Jagannathan. He completed summer internships at the Sun Microsystems Laboratories in 2001 and at the IBM TJ Watson Research Center in 2004.

Adam's work is in the area of programming language design and implementation, with specific interests in developing new concurrency and synchronization models, transaction processing, and compiler and run-time system optimizations.