Advanced Concurrency Control in Java

Pascal A. Felber
Bell Laboratories
600 Mountain Ave, Room 2B-303
Murray Hill, NJ 07974
pascal@research.bell-labs.com

Michael K. Reiter
Bell Laboratories
600 Mountain Ave, Room 2T-316
Murray Hill, NJ 07974
reiter@research.bell-labs.com

ABSTRACT

Developing concurrent applications is not a trivial task. As programs grow larger and become more complex, advanced concurrency control mechanisms are needed to ensure that application consistency is not compromised. Managing mutual exclusion on a per-object basis is not sufficient to guarantee isolation of sets of semantically-related actions. In this paper, we consider "atomic blocks", a simple and lightweight concurrency control paradigm that enables arbitrary blocks of code to access multiple shared objects in isolation. We evaluate various strategies for implementing atomic blocks in Java, in such a way that concurrency control is transparent to the programmer, isolation is preserved, and concurrency is maximized. We discuss these concurrency control strategies and evaluate them in terms of complexity and performance.

1. INTRODUCTION

Writing concurrent programs is a challenging task. While it is well known that shared resources must be protected from concurrent accesses to avoid data corruption, guarding individual resources is often not sufficient. Sets of semantically related actions may need to execute in mutual exclusion to avoid semantic inconsistencies. While databases have native support for such "transactional" constructs, most concurrent programming languages lack adequate mechanisms to handle this task.

The system model and assumptions of concurrent applications are generally different from those of databases: Unlike databases, concurrent programs generally manipulate transient data and may not be able to "undo" a set of actions (rollback). This means that concurrency control mechanisms should avoid situations where rollback is necessary (such as deadlocks), and should implement conflict avoidance rather than conflict resolution. This can translate into the use of pessimistic locking strategies instead of the optimistic strategies often used in databases. Another difference is that the the code of a concurrent application may be arbitrary complex and may not easily be reduced to read and write operations on data items. This is especially true of code that was not developed with concurrency in mind, but is executed a posteriori in a concurrent context.

Concurrency control mechanisms that implement mutual exclusion of multiple actions in concurrent applications face a tradeoff: On the one hand, control over shared resources must be acquired in a conservative way to avoid situations where rollback would be necessary. On the other hand, control over these shared resources must be held for the shortest amount of time possible to increase concurrency. While this tension has been extensively studied in databases [4], surprisingly little work has been performed in the context of concurrent programming languages.

This paper discusses concurrency control mechanisms for implementing atomic sets of actions in Java, a general-purpose, object-oriented concurrent programming language. The goal is provide simple yet efficient mechanisms to implement mutual exclusion on arbitrary sets of objects, in order to increase concurrency of multi-threaded application without violating safety. We take advantage of the object-oriented nature of the language to guarantee isolation in a transparent way and decouple the declaration of critical sections from the underlying mutual exclusion mechanisms. Code executing in an atomic block does not need to be aware of concurrency, and existing applications only require trivial modifications for taking advantage of our mechanisms. Several concurrency control strategies are presented and evaluated in terms of complexity and performance. While the mechanisms discussed in this paper have been packaged as a class library for ease of implementation, they could easily be added to the language through a simple extension of Java's "synchronized" statement.

The rest of the paper is organized as follows. Section 2 introduces background concepts and presents the motivations of this work. Section 3 briefly discusses related work. Section 4 describes the various locking policies supported by our Java concurrency control framework. Section 5 discusses the implementation of atomic blocks in Java using the locking policies previously introduced. Section 6 presents experimental results from our Java implementation, and compares the different policies in terms of concurrency and runtime performance. Finally, Section 7 concludes the paper.

2. BACKGROUND AND MOTIVATIONS

Consider the simple problem of transferring money from one bank account to another.[1] This transfer operation must be atomic, in the sense that any other entity accessing these accounts concurrently will see their balance before or after the transfer, but not in between the withdrawal and the deposit. For instance, a concurrent operation that computes the sum of both bank accounts would return inconsistent results if it sums the balance of bank accounts after the withdrawal but before the deposit: the sum of the balances is a semantic invariant that should not be violated.

Databases have native support for such constructs. They guarantee that operations gathered into transactions satisfy the four so-called ACID properties: *Atomicity*, i.e., transactions executes completely or not at all; *Consistency*, i.e., transactions are a correct transformation of the state; *Isolation*, i.e., even though transactions execute concurrently, it appears for each transaction T that others transactions execute either before T or after T, but not both; and *Durability*, i.e., modifications performed by completed transactions survive failures. Databases implement this behavior by controlling access to shared data, and undoing the actions of a transaction that did not complete successfully (roll-back).

The cost of running a transaction in a database is not negligible, and applications that do not need all four ACID properties could benefit from using more lightweight mechanisms. In this paper we only focus on isolation guarantees for concurrent applications that essentially manipulate tran-

[1] We chose the bank transfer example to illustrate our problem because of its simplicity and intuitiveness. Note however that bank accounts are typical examples of critical data that *should* be persistent and kept in a database.

sient data, do not need durability, and never need to abort (mandating arbitrary actions of a concurrent application to be reversible is incompatible with the goals of keeping concurrency management transparent). Using a database in this context is obviously inadequate.

In our bank application, application consistency can be preserved by making the withdrawal and the deposit part of an *atomic block* that cannot be interrupted by concurrent threads accessing the same bank accounts. In the rest of this paper, we will refer to the set of operations of an atomic block using the generic term of "transactions", even though they are not formally equivalent to *database* transactions that satisfy all four ACID properties. Figure 1 shows how the bank transfer might be implemented in Java if the language had an "atomic" keyword for declaring atomic blocks.

```
class Bank {
  void transfer(Account from, Account to, int amount)
  {
    atomic {
      from.withdraw(amount);
      to.deposit(amount);
    }
  }
}
```

Figure 1: Atomic transfer between bank accounts with an hypothetical "atomic" keyword.

In a programming language that does not natively support transactions, like Java, isolation must be implemented using concurrency control mechanisms. Java's built-in concurrency support [5] allows programmers to create multiple threads and let them execute simultaneously. Each Java object contains a *synchronization lock* which can be used to implement mutual exclusion: only one thread at a time can hold the lock.

Java defines the "synchronized" keyword to acquire the lock of an object and guard a method or a block of code. Synchronized methods acquire the lock of the target object or class for the duration of the method. The more versatile synchronized block construct locks an arbitrary Java object for the duration of the block. However, it is not possible to atomically acquire the locks of multiple objects for a synchronized block.

```
1  class Bank {
2    synchronized
3    void transfer(Account from, Account to, int amount)
4    {
5      from.withdraw(amount);
6      to.deposit(amount);
7    }
8  }
9  // Thread 1:
10 bank.transfer(a1, a2, 1000);
11 // Thread 2:
12 bank.transfer(a3, a4, 2000);
13 // Thread 1 and thread 2 are serialized
```

Figure 2: Synchronizing on a global object reduces concurrency.

A first solution to the bank transfer problem using a synchronized method is given in Figure 2. The synchronization lock of the bank object is acquired when entering the "transfer" method and released upon completion, thus ensuring that no two threads can execute this method concurrently. The problem with this approach is that it does not discriminate between transfers that interfere and those that do not. For instance, the two concurrent transfers shown in the figure (lines 10 and 12) will be serialized, although they do not access the same accounts and thus do not interfere. If the bank manages a large number of account and interferences are not frequent, this approach is obviously inadequate: it guarantees isolation but significantly limits concurrency.

Another solution is given in Figure 3. Instead of obtaining the lock on the bank, we obtain the locks on all account objects part of the transaction. This is implemented using nested synchronized blocks (lines 4–5). The major problem of this solution is that it introduces risks of deadlock. A deadlock is a form of *liveness* interference in that it prevents progress. As shown in the figure, two threads

```
1  class Bank {
2    void transfer(Account from, Account to, int amount)
3    {
4      synchronized(from) {
5        synchronized(to) {
6          from.withdraw(amount);
7          to.deposit(amount);
8        }
9      }
10   }
11 }
12 // Thread 1:
13 bank.transfer(a1, a2, 1000);
14 // Thread 2:
15 bank.transfer(a2, a1, 2000);
16 // Thread 1 and thread 2 may deadlock
```

Figure 3: Nested synchronized blocks may cause deadlocks.

performing concurrent transfers on the same accounts but in the reverse order may deadlock (lines 13–15). Indeed, if one thread locks the first account at the same time as the other thread locks the second account, we run into a deadlock situation because each thread will try to acquire a lock held by the other thread, thus violating liveness. Database systems traditionally solve deadlocks by selectively aborting some transactions. In a concurrent program, it is generally not possible to detect deadlocks and/or abort transactions, and the appropriate strategy is to avoid deadlock.

Another problem of this approach is that it cannot easily be applied to an arbitrary number of objects (not known statically). For instance, it is not straightforward to implement a method that takes an array of bank accounts and compute the sum of their balances, because the number of nested synchronized blocks depends on the number of accounts, which is not known at compile time.[2] The limitations of Java's concurrency control mechanisms for transactional operation are further discussed in [5].

The main motivation of this work is to provide generic mechanisms to solve these kinds of problems. Isolation mechanisms should have minimal impact on the application's code (non-intrusiveness) and should increase concurrency while avoiding deadlocks, i.e., provide both liveness and safety.

[2]A practical solution to this problem is to use recursion to simulate an arbitrary number of nested synchronized blocks. However, this solution is complex and lacks generality.

3. RELATED WORK

There exist numerous languages or libraries for parallel programming with various levels of transactional support (see [11] for a survey). They introduce high-level tools and paradigms adapted to the development of parallel applications, by enabling the decomposition of complex programs into multiple tasks that can execute concurrently on parallel or distributed architectures. When available, transactional semantics are generally implemented through distributed commit protocols.

In contrast, general-purpose programming languages with multi-threading support (such as Java) generally provide low-level concurrency-control mechanisms like locks, semaphores, or monitors that guarantee mutual exclusion to specific sections of code [7]. While flexible, these mechanisms are not well adapted to non-trivial problems such as isolation of multiple concurrent transactions.

For efficiency reasons, database management systems (DBMSs) generally implement advanced concurrency control mechanisms for executing numerous transactions concurrently while guaranteeing ACID properties [4]. DBMSs focus on *persistent* data management and provide no or limited concurrency control mechanisms for code executing outside of the DBMS.

The mechanisms presented in this paper have a different, less ambitious goal than parallel programming languages or DBMSs. Instead of defining new tools and paradigms for parallel programming or transaction management, our goal is to provide a few simple, transparent mechanisms for increasing concurrency of Java applications while preserving some limited form of transactional integrity. These mechanisms can be easily added to existing applications, without the need of a specialized programming language or deployment of the application's data in a DBMS.

Java already offers two transaction frameworks: the *Java Transaction API (JTA)*, part of the enterprise edition of the Java platform (J2EE) [9], and *Jini Transactions* [12]. The Java Transaction API is a set of local interfaces between a transaction manager and the parties involved in a distributed transaction system: the application, the resource manager, and the application server. It includes transactional application interfaces, a Java mapping to the standard X/Open XA protocol, and a transaction manager interface.

While JTA aims at providing a complete set of transactional mechanisms to Java applications, the Jini Transaction Specification provides a minimal set of protocols and interfaces to allow objects to implement transactional semantics. The responsibility of actually implementing these semantics is left to the individual objects that take part in a transaction. Coordination between transaction objects is achieved through a *two-phase commit* protocol, which is the most widely used protocol for distributed transactions.

Both Java transaction frameworks differ from the work presented in the paper by several aspects. First, both JTA and Jini transactions essentially target *distributed* transactions, (1) as APIs to a complete distributed transaction system or (2) as minimal interfaces for distributed coordination between transactional Java objects. A consequence of distribution is that these frameworks must deal with situations where transactions abort because of exceptional conditions that affect only some of the distributed components (such as partial failures or local scheduling conflicts). Finally, JTA and Jini transactions essentially provide a declarative APIs to the basic components of a transactional system and thus require transaction participant to support specific interfaces and take part to well-defined protocols. In contrast, the work presented in this paper is more restrictive in that it does not deal with distributed transactions, it does not guarantee transaction durability nor allows transactions to abort, and it focuses on providing transparent integration of transactional facilities into the programming language rather than through a programmatic API.

4. LOCKING POLICIES

To ensure mutual exclusion on a set of shared resources, threads must lock these resources prior to accessing them, and release the locks when they are no longer needed. The strategy used for acquiring and releasing locks is called the *locking policy*. Locking policies try to maximize concurrency by minimizing the time during which locks are held. In this paper, we only consider locking policies that avoid deadlocks and thus do not require undoing partial transaction execution.

In this section, we present several locking policies that offer various tradeoffs in terms of overhead, concurrency, and required transaction knowledge. A good understanding of these policies is important for maximizing the performance of a concurrent application. The first few policies are variations of so-called *two-phase locking* (2PL) strategies [1], while the last one is a non-2PL policy. Our Java implementation of atomic blocks can use any of these policies.

To illustrate these locking policies, we consider the following simple example that involves three transactions T_1, T_2, and T_3 executed concurrently on four objects a, b, c, and d (Figure 4). Unlike typical database transactions, we do not distinguish between read and write operations: we assume that each object has a set of operations ("op" in the figure) that can perform arbitrary accesses to the state of the object.

$$\begin{array}{lcl} T_1 & : & a.op()\ ;\ b.op()\ ;\ c.op()\ ;\ d.op() \\ T_2 & : & a.op()\ ;\ b.op() \\ T_3 & : & c.op()\ ;\ d.op() \end{array}$$

Figure 4: Three sample transactions.

4.1 Two-phase Locking

The best-known deadlock-free locking policy is *two-phase locking* (2PL). All objects accessed by a transaction are locked during the first phase and released during the second phase. It is not possible to unlock an object before all objects have been locked, or to lock an object once any lock has been released. There exist several variations of 2PL protocols, some of which are discussed in the rest of this section.

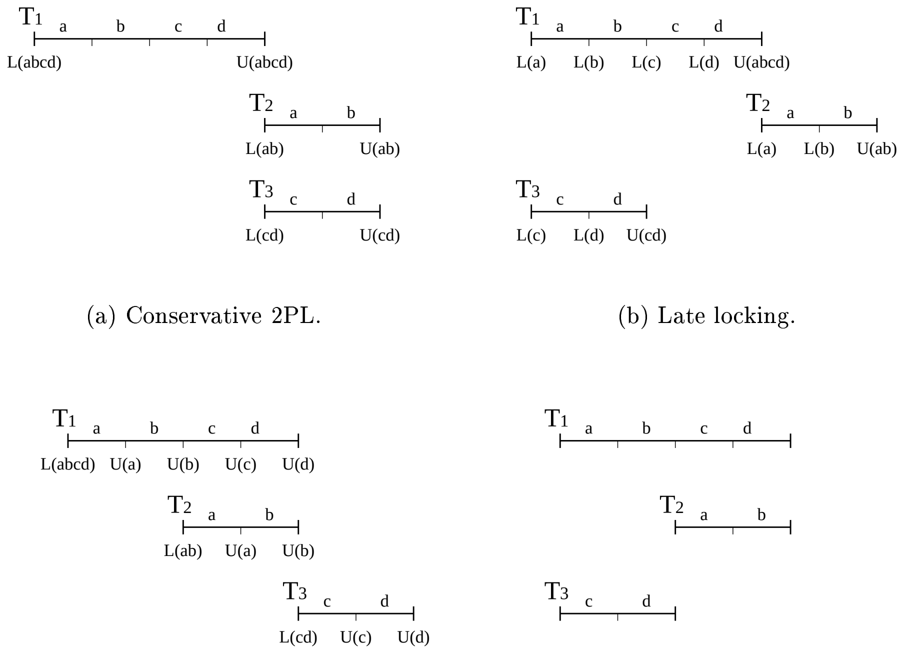
In order to avoid deadlocks, objects should be locked in an order consistent with a total order on the objects. We assume that there exists a unique value $\#o$ associated with each object o that can be used to assign ranks to objects. Objects are always locked in increasing rank order, thus avoiding deadlocks (the order in which resources are unlocked does not matter). In our example, we assume that $\#a < \#b < \#c < \#d$.

Conservative 2PL. The most basic 2PL protocol is *conservative 2PL* (also known as *static 2PL*). With this protocol, all objects are locked before starting the transaction, and unlocked after the transaction has completed. Operations of the transaction execute only when *all* objects are locked.

Figure 5(a) shows an execution history of the transactions of Figure 4 with a conservative 2PL policy. A transaction is represented by a horizontal line, split into multiple segments that represent individual operations. We indicate above each operation the object accessed by that operation. Lock acquisition and release are represented in the figures using the notation $L(o)$ for locking an object o and $U(o)$ for unlocking o. We consider that each individual operation consumes one unit of time and successful locking and unlocking takes no time. Therefore, execution of all three transactions take 6 units of time.

2PL with Late Locking. A first optimization to conservative 2PL is to wait until an object is actually accessed for locking it. This technique, known as *strict 2PL* in the database world, will be referred to as *2PL with late locking* in this paper.

As with conservative 2PL, objects are locked in increasing rank order to avoid deadlocks. Therefore, the late locking protocol works as follows (Figure 5(b)). Before accessing an object o, the transaction T checks if o is already locked. If it is not the case, T locks every object o' accessed by T such that $\#o' \leq \#o$ and o' is not yet locked, in increasing rank order. Therefore, the effectiveness of this policy strongly depends on the order in which objects are accessed. If objects are mostly accessed in the same order as their rank, then the late locking policy can significantly increase concurrency



(a) Conservative 2PL. (b) Late locking.

(c) Early unlocking. (d) Optimal locking.

Figure 5: Execution of the transactions of Figure 4 with various locking strategies.

over conservative 2PL (if we only consider execution of T_1 and T_2, execution completes in 4 units of time vs. 6 for conservative 2PL). On the other hand, if the first object accessed by a transaction is the object with the highest rank, then late locking is equivalent to conservative 2PL.

2PL with Early Unlocking. *2PL with early unlocking* is another variation of 2PL. However, unlike late locking, the effectiveness of early unlocking does not directly depend on the order in which objects are accessed.

With early unlocking, all objects accessed by a transaction are locked at the beginning of the transaction. After each operation, we check if the object accessed by the last operation will be accessed again by the transaction. If this is not the case, we release the lock on that object. In other words, objects are locked from the begin of the transaction up to the last operation that accesses them.

Early unlocking generally achieves better concurrency than conservative 2PL. For instance, if we consider only transactions T_1 and T_2, early unlocking executes in 4 units of time (vs. 6 for conservative 2PL). For a given set of transactions, each of the late locking and early unlocking strategies can have the edge. For instance, late locking performs better with T_1 and T_3 while early unlocking is more efficient with T_1 and T_2. Late locking generally provides slightly lower concurrency with random transactions that early unlocking because it requires objects to be accessed in the same order as they are locked to perform optimally. On the other hand, the early unlocking protocol needs to know when an object is no longer needed in the transaction, i.e., the application must provide a description of the transaction for taking advantage of early unlocking.

Generalized 2PL. The last flavor of 2PL discussed in this paper is *generalized 2PL*. It combines the optimizations of late locking and early unlocking. Locks can be acquired late and released early as long as the locking pattern complies with the basic 2PL protocol.

In theory, there exist multiple lock acquisition patterns for a given transaction with generalized 2PL. Some of these patterns are more efficient than others, but choosing the best pattern requires "global" knowledge of the transactions executing in the system. For instance, with the transactions of Figure 4, generalized 2PL can execute all transactions in 4 units of time if it executes transaction T_1 according to the following schedule: $L(a)$; $L(b)$; $a.op$; $b.op$; $L(c)$; $L(d)$; $U(a)$; $U(b)$; $c.op$; $d.op$; $U(c)$; $U(d)$. The choice of locking c and d and unlocking a and b between the second and third operations of T_1 is arbitrary and may be motivated by static transaction knowledge or runtime heuristics.

In practice however, a generalized 2PL protocol tries to acquire locks as late as possible and, when all locks have been obtained, releases them soon as they are no longer needed. We call this protocol "deterministic" generalized 2PL because the lock acquisition pattern does not depend on other factors than the structure of the transaction on which it is applied. In the rest of this paper, we will only consider this variant of generalized 2PL. With the transactions of Figure 4, deterministic generalized 2PL executes transaction T_1 according to the following schedule: $L(a)$; $a.op$; $L(b)$; $b.op$; $L(c)$; $c.op$; $L(d)$; $U(a)$; $U(b)$; $U(c)$; $d.op$; $U(d)$. This schedule is almost equivalent to late locking and executes in 5 units of time.

4.2 Tree Locking

The deadlock-free 2PL locking policies have in common that no object can be unlocked before all objects have been locked, and objects must be locked in a predefined order. *Tree locking* [10] is a non-2PL policy that avoids these limitations by using different rules to decide when and in which order to lock and unlock objects. Tree locking is a deterministic, deadlock-free locking policy that is optimal for our example: it executes all three transactions in 4 units of time, as shown in Figure 5(d) (lock acquisition and release are not shown in the figure and will be discussed after the tree locking protocol has been introduced).

Tree locking was originally developed to take advantage of the hierarchical structure of a database, represented as a tree. Transactions always access data items by following paths in the tree. Any node in the tree can be locked, and locks held on a node implicitly propagate to all of its children. A transaction starts by locking[3] the top-most node of the tree. Then, it travels down to the data item to be accessed, locking every intermediate node. A node N can be unlocked when the transaction has obtained all the locks it needs on N's children. Once unlocked, a node cannot be locked again. A direct consequence of this protocol is that the order in which locks are obtained depends on the the structure of the tree, not on an order relation between individual data items.

To increase concurrency of atomic actions in concurrent applications, we use a variation of the tree locking protocol used in databases. Resources are organized in a tree: data

[3]For simplification we assume that there is only one type of lock.

items (i.e., shared objects) are located on leaves of the trees, and internal nodes are "artificial" objects that impose relationships between resources and coordinate lock acquisition and release. Since internal nodes are not data items, the tree does not depend on the physical structure of the data and can dynamically evolve into configurations that are optimal for the transactions being processed. Details of the tree locking protocol are given in Appendix A.

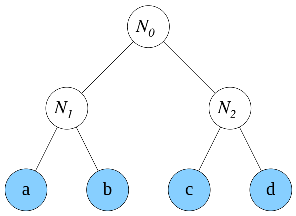


Figure 6: With tree locking, shared resources are organized in a tree.

The tree locking protocol with the tree of Figure 6 results in optimal execution for the transactions of Figure 4. It takes only 4 units of time, which is the length of the longest transaction, and there are always two transactions executing concurrently. Tree locking has however the same drawback as early unlocking: the protocol needs to know when an object is no longer needed in the transaction. In addition, the runtime overhead of tree locking is the biggest among all protocols presented in this paper, since more locks need to be acquired and released. Indeed, transactions need to lock the nodes of the tree, in addition to the data items actually accessed.

4.3 On Performance and Concurrency

In this section, we have presented several 2PL locking protocols, as well as a non-2PL tree locking protocol. Each locking protocol has benefits and drawbacks. A general rule is that complex protocols have more runtime overhead but potentially achieve increased concurrency. Although we will discuss performance in Section 6, we present a few preliminary observations below.

First, when there is low contention (i.e., it happens rarely that two transactions compete to access a shared object at the same time), policies that have small runtime overhead perform better. In this scenario, conservative 2PL is generally the best choice.

On the other hand, when there is much contention it is important to maximize concurrency, even at the price of additional runtime overhead. In these situations, a locking policy like generalized 2PL or tree locking is more adequate. Experiments show that 2PL policies permit significantly more concurrency than tree locking with a static tree and random transactions. However, with a tree that is "adequate" for a set of transactions (i.e., the structure of the tree is optimized for these transactions), tree locking can increase concurrency substantially over 2PL protocols. In particular, tree locking appears to be a promising approach when working with structured data.

The problem of finding a tree that is adequate for a given set of transaction is not trivial. We have identified four adequacy criteria that characterize a good tree for a given set of transactions (see Appendix B): (1) The root node of a transaction should be as deep in the tree as possible. (2) The acquisition of a node must pay off and concurrency can be optimal when transactions access all resources located below that node. (3) Concurrency is increased if accesses to the resources of a subtree are adjacent in a transaction. (4) Concurrency is generally increased if shared resources are accessed by multiple transactions in the same order.

When data is naturally organized in a hierarchical manner and accesses follow structured patterns (e.g., traversal of a sub-tree), then a good tree can be trivially inferred from the data's hierarchical structure. On the other hand, when data and accesses are not structured, finding a tree that is adequate for a given set of transaction is a difficult problem. This problem can be stated as follows:

Consider a set of n transactions $T_1, ..., T_n$ with sizes $m_1, ..., m_n$. Each transaction T_i is composed of m_i individual operations $o^i_1, ..., o^i_{m_i}$ and is executed by a different thread. All threads start at the same time. Individual operation all take one unit of time, and concurrency management operations (lock acquisition and release) happen instantaneously. When two transactions try to acquire a lock at the same time, the lock is granted to the transaction with the smallest index (i.e., T_i will acquire the lock before T_j if $i < j$).

PROBLEM 4.1. *Given a set of n transactions $T_1, ..., T_n$, find (1) a tree F_{min} such that all transactions complete in minimum time, and (2) a tree F_{avg} such that the average time required by each transaction to complete is minimum.*

The tree F_{min} is optimal for a one-time execution of the transactions, while the tree F_{avg} is better when each thread executes more than one transaction. This problem can be shown to be NP-hard (see Appendix C). As a result, we have primarily focused on heuristics for building a *good* tree in polynomial time. To evaluate the effectiveness of tree locking with unstructured data and transactions, we have implemented a simple greedy algorithm that produces balanced binary trees where objects are organized according to their frequency and proximity in the transactions. This algorithm tries to place objects that are close in the given transactions in the same subtree, with the priority given to objects that are accessed more often. The details of the algorithm are given in Appendix D. Experiments results with tree locking and the tree construction algorithm are discussed in Section 6.

5. ATOMIC BLOCKS IN JAVA

This section describes the implementation of atomic blocks in our Java Concurrency Framework (*JCF*). We first present the design goals and introduce the notions of atomic object and atomic block. We then describe the various mechanisms used for providing transparent concurrency management and discuss the benefits and drawbacks of each of them. For ease of implementation, these mechanisms have been packaged as a set of Java classes; we do however believe that basic support for atomic blocks would be a desirable extension to the Java language, as proposed in the end of this section.

5.1 Goals

Implementation of atomic blocks in *JCF* was influenced by the following design goals:

- **Transparency.** Code should not be modified for executing within an atomic block.
- **Generality.** Atomic blocks can be placed around arbitrary Java code.
- **Efficiency.** Atomic blocks should add as little runtime overhead as possible while maximizing concurrency.
- **Separation of concerns.** The declaration of an atomic block should be independent of the locking strategy.

The first goal — transparency — states that atomic blocks should not be visible by code executing within the block and should not require modifications to that code. This also means that legacy code, written without concurrency in mind, can execute safely in a concurrent environment just by surrounding critical operations with atomic block constructs.

The second goal — generality — requires support for arbitrary code inside atomic blocks, as within a "synchronized" statement. This code can perform arbitrary operations and use any language construct, as long as it executes in the context of a single thread of control. Transactions do *not* need to be described in a separate language, such as SQL, for managing concurrency and maintaining consistency.

The third goal — efficiency — means informally that the runtime overhead of concurrency control mechanisms should not be higher than the performance improvements resulting from increased concurrency. On the one hand, serial execution can be implemented with very low runtime overhead, but no effective concurrency. On the other hand, advanced concurrency control mechanisms have higher runtime overhead, but also better concurrency. Atomic blocks should try to minimize runtime overhead and maximize concurrency.

The last goal — separation of concerns — states that the locking strategy used for ensuring isolation of atomic blocks should be independent of the atomic block declaration. In other words, the application developer can declare an atomic block without having to know how concurrency control is implemented, and the system developer can program a locking strategy for atomic blocks without having to know the application's code. It follows that it must be possible to configure the locking strategy at deployment time (or even at runtime) without changes to the application's code.

Note that JCF does not aim at being a full transaction framework, intended to replace a DBMS. It rather focuses on transparent mechanisms to ensure isolation and atomicity of concurrent object invocations and seamless integration with programming language constructs. A consequence of our transparency goals is that we do not distinguish between read and write operations and we consider a restricted transaction model that does not guarantee durability and does not allow transactions to abort (no rollback). JFC can be used for instance to maintain consistency of in-memory data structures (e.g., B-tree, XML data tree) accessed by multiple threads. Such data does not need to be persistent, but its complex structure and large size can making explicit concurrency control error-prone and subject to poor performance. JCF hides this complexity by allowing non-trivial operations such as moving data or traversing subtrees to be performed concurrently on arbitrary nodes without having to explicitly deal with concurrency control.

5.2 Atomic Objects and Atomic Blocks

An *atomic object* [6] is an object that can be accessed concurrently by several threads. Even though accesses are concurrent, an atomic object behaves as if accesses occur one at a time, in an order which is consistent with the order of invocations and responses. The smallest granularity of atomicity supported by *JCF* is the invocation of an atomic object. *JCF* also provides concurrency control mechanisms that guarantee isolation of sequences of invocations on atomic objects. Such a sequence of invocations forms an *atomic block*.

An atomic object is essentially an application-specific object whose concurrency is managed by *JCF*. Application can render an arbitrary object atomic by calling a *JCF*-specific method (this is a one-time procedure performed during application initialization). If the application object does not already behave like an atomic object (i.e., it does not support concurrent invocations), *JCF* transparently serializes invocations to that object. This guarantees that objects remain consistent individually. Global (or transactional) consistency is maintained using atomic blocks.

An atomic block executes sequences of invocations to atomic objects (and other instructions) in isolation. It is instantiated with the set of atomic objects that it manages as a parameter, and is semantically bound to a thread of control. Atomic blocks can be arbitrarily nested in practice, but in that case — similarly to "synchronized" statements — there exists a risk of deadlock. Atomic blocks provides two methods, "begin" and "end", that act as delimiters. The code executing between these methods executes in isolation of other atomic blocks. Atomic blocks are represented by objects that implement the "AtomicBlock" interface. There are several kinds of atomic blocks that implement different locking policies.

```
1  class Bank {
2    void transfer(Account from, Account to, int amount)
3    {
4      AtomicBlock ab;
5      ab = Atomic.newAtomicBlock(new Object[] {from, to});
6      ab.begin();
7        from.withdraw(amount);
8        to.deposit(amount);
9      ab.end();
10   }
11 }
12 // Initialization
13 for(int i = 0; i < accounts.length; i++)
14   accounts[i] = (Account)Atomic.makeAtomic(accounts[i]);
15
16 // Thread 1:
17 bank.transfer(accounts[0], accounts[1], 1000);
18 // Thread 2:
19 bank.transfer(accounts[1], accounts[2], 2000);
20 // Threads 1 and 2 conflict and execute in isolation
21
22 // Thread 3:
23 bank.transfer(accounts[3], accounts[4], 1500);
24 // Thread 3 executes concurrently with treads 1 and 2
```

Figure 7: Atomic blocks improve concurrency while ensuring isolation.

Figure 7 shows an implementation of the bank application of Section 2 that uses atomic blocks. Initially, all account objects are made atomic (lines 13–14). In the transfer method, an atomic block is instantiated with the source and destination account as parameter (line 5). The money transfer is performed inside the atomic block (lines 7–8), delimited by the invocations to "begin" and "end" on the atomic block object (lines 6 and 9). The runtime concurrency control mechanisms ensure that the first and second transfers (lines 17 and 19), which conflict, execute in isolation. The third transfer (line 23), which does not conflict with the other transfers, can execute concurrently. Although not shown in the code, a good practice is to include the instructions of an atomic block in a "try-catch" statement and end the atomic block in the "finally" block. This ensures that all resources and locks acquired by the concurrency control protocol will be released when exiting the atomic block.

Atomic blocks can be customized in several ways (via overloading of the "newAtomicBlock" method). In particular, they are optionally parameterized by a locking policy, which can be chosen at runtime (some guidelines for selecting a locking policy are given in Subsection 5.4). In the case of tree locking, the programmer can also provide a tree generator, whose function is to construct a tree adequate for the given transactions. Trees can evolve over time, and it is possible to use different trees for non-intersecting sets of objects. For locking policies that require a description of the transactions' structure (tree locking and 2PL policies that implement early unlocking), atomic blocks are further parameterized by a "Transaction" object, which enumerates the individual operations of the transaction and the objects they access.

5.3 Intercepting Invocations

As previously stated, a major goal of atomic blocks is to manage arbitrary code, without having to perform modifications to that code. A direct consequence is that the *JCF* runtime must be able to *transparently* perform concurrency control operations during execution of an atomic block. Indeed, all locking policies discussed in this paper except conservative 2PL acquire and release locks in the middle of atomic blocks, immediately before or after invocations to atomic objects.

JCF performs dynamic concurrency control management by intercepting invocations to atomic objects. As part of the process through which objects are made atomic, *JCF* transparently encapsulates the application object inside a system-level wrapper that can pre- and post-process any request targeted to the application object. Among the operations performed by this wrapper are object atomicity (if an application object is not atomic, the wrapper serializes invocations to that object) and block isolation (lock acquisition and release according to the atomic block's locking policy).

JCF performs the actual interception of invocations through the well-known technique of *object proxying*. A proxy is and object that acts as a surrogate or delegate for another object, and usually behaves in such a way that the its invokers have no indication that they deal with a proxy instead of the underlying object being proxied (see the *proxy design pattern* in [3]). Object proxying is implemented in *JCF* using one of three approaches: dynamic proxies, static proxy generation, and custom proxies. These approaches are described in the rest of this section.

Dynamic Proxies. Dynamic proxies are a mechanism introduced in Java 1.3, which permit the creation of a class that implement a set of interfaces specified at runtime [2]. A dynamic proxy object receives all invocation targeted at the proxied object(s) and can perform arbitrary tasks instead of, prior to, or after forwarding the request to its actual target.

JCF's dynamic proxy implementation permits registration of per- and post-invocation handlers. Each locking protocol provides its own invocations handlers, which are registered upon entering an atomic block and unregistered at its end. Various locking protocols have different needs in terms of invocation handlers: conservative 2PL does not use invocation handlers, 2PL with late locking only uses pre-invocation handlers, 2PL with early unlocking only uses post-invocation handlers, and generalized 2PL and tree locking use both. In addition to pre- and post-invocation handlers, dynamic proxies also ensure object atomicity.

Dynamic proxies have three drawbacks. First, they are a recent addition to the Java language and are not widely deployed yet. Second, because of their dynamic nature, they have a non-negligible runtime overhead. Indeed, dynamic proxies intercept and process requests using Java's reflection API, which has a high cost in terms of performance. Finally, dynamic proxies only intercept operations declared on interfaces. In other words, for using dynamic proxies, all operations of the application object must be declared in interfaces implemented by that object.

Static Proxy Generation. The second approach for intercepting invocations consists is generating static proxies for atomic objects. A static proxy implements the same methods as the actual object. Each method of the static proxy performs three operations: pre-processing, invocation to the actual object, and post-processing. During pre- and post-processing, the static proxy performs the same concurrency control operations as dynamic proxies. The actual processing of the request is delegated to the target object through a static method call.

The static proxy generator uses reflection to discover the methods implemented by application objects. Proxy generation can happen at compile-time or at runtime. In the first case, the code of the proxy is generated in a file that must be compiled to produce the proxy class. In the second case, the proxy is directly generated as bytecode and dynamically loaded in memory by the Java class loading mechanisms. The latter approach is more convenient because the developer does not need to deal with proxy classes. It does however require runtime permissions that may not be granted to code executing in a protected environment, such as applets.

Since static proxies intercept and invoke operations on application objects statically, their runtime overhead is significantly smaller than dynamic proxies. Static proxies also do not suffer from the same limitations as dynamic proxies, which only intercept invocations to the methods declared on the interfaces implemented by an object.

Custom Proxies. *JCF* provides a third approach to intercept invocations, in which the developer can explicitly control how concurrency control is applied to application objects. With this method, the programmer is responsible for ensuring atomicity of objects, and for calling *JCF* pre- and post-invocation handlers at relevant places in the code (concurrency control is explicitly delegated to *JCF*).

Custom proxies are the most flexible approach, because the programmer can control when and how concurrency control is applied to application objects. This may lead to fine-grain optimizations, such as disabling concurrency control for methods that are not required to execute in isolation. On the other hand, custom proxies are also the most "dangerous" approach because the programmer has to comply with a set of rules that, if not followed, may lead to violations of transaction isolation or deadlocks. In addition, it requires code modifications, which makes its application to legacy code less straightforward.

5.4 Design and Runtime Choices

JCF is a versatile concurrency framework that offers a variety of choices. The locking strategy influences the concurrency degree of the application, and the interception mechanisms affects the runtime overhead and in some respects the programming model.

Decisions about the locking strategy can be performed late in the development cycle, as late as at runtime. It is possible to use multiple locking policies in the same application, with the following restrictions. All 2PL policies are compatible with each other and any combination of these policies can be used simultaneously in an application. Tree locking and 2PL are not compatible and they should not be used to manage the same resources. When using tree locking for a set of objects, all threads that access these objects concurrently must use the same tree to guarantee isolation. This is enforced by *JCF* runtime, which does not allow an object to be part of multiple trees.

The locking policy should be chosen to yield the best performance for the application. The experimental results presented in Section 6 can give guidelines on how locking strategies behave with some type of applications. If the application exhibits repeatable access patterns, it may be wise to test each locking strategy and chose the most efficient prior to deploying the application.

Unlike with locking policies, the criteria for selecting an interception mechanism are not only based on performance. Transparency and security constraints are other factors that can influence this choice. Dynamic proxies require almost no modifications to legacy application but are limited to proxying interfaces and add significant runtime overhead. Static proxies are more efficient and powerful, but they can be cumbersome to manage and require additional permissions in the case of runtime proxy generation. Custom proxies are the most flexible approach, but it requires the programmer to perform substantial modification to his/her code. The runtime impact of the different interception mechanisms is discussed in greater detail in Section 6. Note that all three approaches are compatible with each other: objects that use different interception mechanisms can coexist in the same application.

5.5 Atomic Blocks as an Extension to the Java Language

The Java language defines a "synchronized" statement that locks an individual object for the duration of the associated block. A simple extension to support atomic blocks in the Java language would be to allow multiple objects as argument of the "synchronized" statement. Without offering the whole spectrum of concurrency control strategies discussed in this paper, the virtual machine could use a conservative 2PL policy to lock all objects in a deadlock-free manner. Since conservative 2PL does not need to know the structure of the transactions in advance, nor does it need to acquire and release locks during execution of the atomic block, no additional modifications should be performed to the syntax and semantics of the "synchronized" statement. In contrast, support for other locking strategies would require additional information to be provided to the Java runtime, e.g., using a thread-specific interface or extra arguments to the "synchronized" statement.

6. EXPERIMENTAL RESULTS

This section presents experimental results with *JCF* and the locking policies described in this paper. We also quantify and discuss the runtime overhead of the different interception mechanisms presented in Section 5.

6.1 The Model

For concurrency measurements, we assume that the actions of locking and unlocking an object take a negligible amount of time. This assumption is realistic with applications where operations that execute in mutual exclusion are time consuming (e.g., disk access, remote invocation, complex computations). The goal of these experiments is not to provide absolute performance figures, but rather to measure the degree of concurrency of an application relative to a serial version of the same application.

For runtime overhead measurements, we concentrate on the cost of concurrency management and interception mechanisms. For this purpose, we ran transactions with operations that do not perform any actual processing (empty operations). All tests have been performed with Java 1.3 on a single-processor PC (P3/750) running Windows NT 4.0.

We have implemented a simulation environment to compare the different concurrency control strategies. The test environment permits the specification of the number of concurrent threads, the length of transactions, the number of objects in the system, the duration of operations, etc. Time consuming operations are simulated by yielding the processor to other threads for a given amount of time (as an I/O operation would do, for instance). The transactions are chosen randomly, but the same transactions are used for all concurrency control strategies. In the tests below, we only used binary trees for tree locking.

6.2 Low Contention Tests

We first consider the case of applications where contention is low, i.e., conflicts are infrequent. For instance, in the example of the bank application, transactions typically have few operations (two for transfers) and the number of accounts is much larger than the number of concurrent transactions, thus leading to low contention.

We have run tests with 32 concurrent transactions, each composed of 2 randomly-chosen operations, on a set of object of variable size. As the number of object grows, contention decreases. The experimental results are shown in Figure 8. The ordinate shows the concurrency degree expressed in percentage with respect to serial execution (i.e., in the case where there is no effective concurrency).

As one can see on the figure, all 2PL locking policies perform well and the concurrency degree approaches the theoretical optimum (3200%) as the number of object grows and contention decreases. There is only little gain from using more elaborate 2PL strategies (e.g., generalized 2PL) over strict 2PL.

On the other hand, tree locking performs poorly and re-

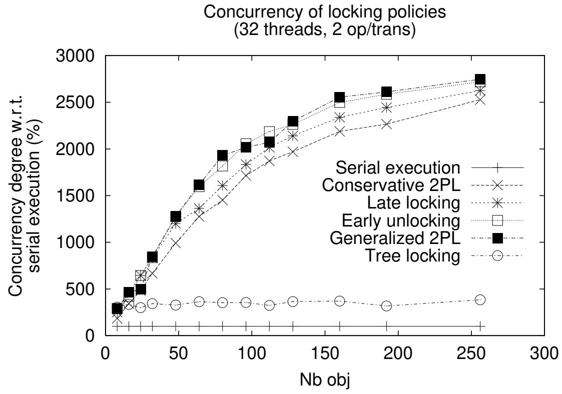


Figure 8: Concurrency degree with low contention tests.

mains almost constant as the number of object grows.[4] This is due to the fact that, with random transaction, there is a 50% likelihood that a transaction with two operations accesses objects located in different halves of the tree, and contention appears on the root and intermediary nodes of the tree rather than on the actual object being accessed. This example demonstrates that tree locking is not suitable for random transactions.

6.3 High Contention Tests

In situations where a large number of threads compete for a small number of resources, contention is high. This may be the case with resources such as files, I/O devices (disks, printers, network interfaces), or more generally application objects that have a large granularity (e.g., a bank object instead of an individual account). The nature of such applications strongly limits the concurrency degree and, as

[4] Figures 8 and 9 show the performance of tree locking with "non-optimized" trees, i.e., without using our algorithm for construction good trees: there was no noticeable improvement when running these experiments with optimized trees, because of the random nature of the transactions.

contention grows, we can expect only little gain over serial execution.

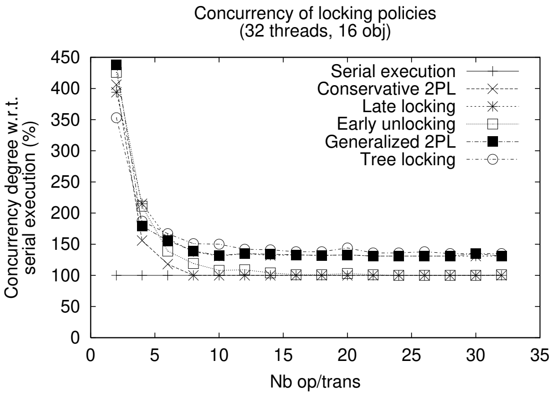


Figure 9: Concurrency degree with high contention tests.

Figure 9 shows execution of 32 concurrent transactions, each composed of variable number of randomly-chosen operations, on a set of 16 objects. As one can see on the figure, as the number of operation per transaction grows and contention increases, the concurrency degree approaches a constant value approximatively 1.5 times better than serial execution. Conservative 2PL and early unlocking even show no gain over serial execution starting from 8 (resp. 16) operations per transaction. Tree locking performs empirically better than 2PL locking policies when contention is high. However, the difference may not be significant enough to justify the use of tree locking over a 2PL policy.

6.4 Hierarchical Data Tests

In situations where data can be organized in a hierarchy, it is straightforward to build a tree that matches this hierarchy and is adapted to tree locking. For instance, XML data can be naturally stored as a tree. Let a "subtree transaction" be a transaction that accesses every object of some subtree exactly once. We have performed tests with 32 concurrent subtree transactions on a variable set of objects. Since we only consider balanced binary trees, the number of objects in the tree is always a power of 2. In addition, because each transaction accesses all the objects of sub-tree (set), transactions also have a length equal to a power of 2. The subtree accessed by each transaction is chosen randomly.

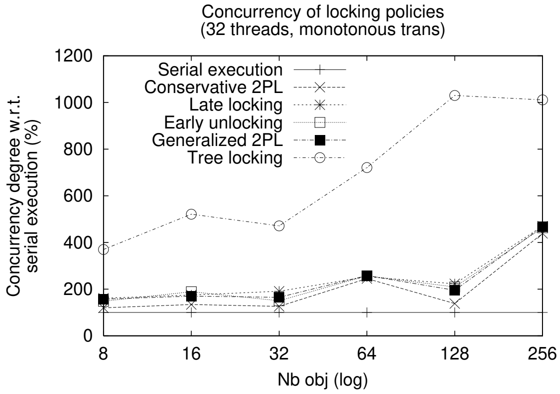


Figure 10: Concurrency degree with hierarchical data.

Figure 10 shows that, with subtree transactions tree locking performs as much as 5 or 6 times better than 2PL locking policies. This can be explained by the fact that, since the structure of the tree matches the access patterns of transactions, many transactions that conflict can still execute concurrently with tree locking. Therefore, the nature of an application and the access pattern of its transactions have a strong impact on the effectiveness of locking strategies and are the key factor for choosing the best strategy.

6.5 Tree Construction Algorithm

When data accesses in a set of transactions are purely random, we noticed that tree locking does not perform well, independent of the structure of the locking tree. We also showed that for hierarchical data and structured accesses, tree locking can significantly increase concurrency. We performed additional experiments to test the effectiveness of the simple tree construction algorithm presented in this paper. For this purpose, we have generated "skewed" transactions, where the objects accessed are chosen according to a Zipf distribution [13]. Some objects are accessed much more often than others, making it important to locate these objects close to each other. For this experiment, we have used short transactions and a variable number of threads.

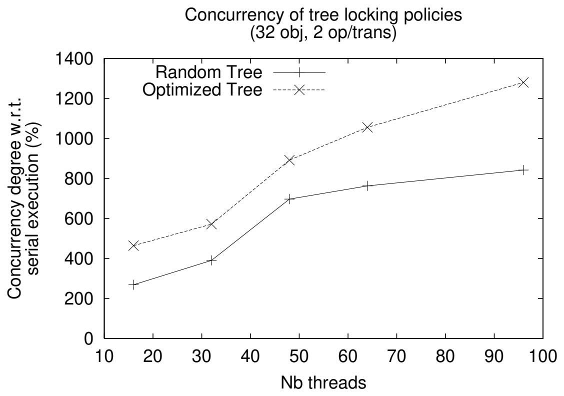


Figure 11: Concurrency degree with skewed transactions (random tree vs. optimized tree).

The results (Figure 11) show significant improvement with the optimized tree, even though the tree generated by the algorithm is sub-optimal. Since real-world applications do not generally access objects at random but according to repeatable patterns, algorithms for generating locking trees adapted to these pattern could be a promising approach for increasing concurrency of those applications.

6.6 Runtime Overhead

In this subsection, we compare the runtime overhead of the various locking policies and the different interception mechanisms. For this purpose, we have run experiments with a single thread that executes a sequence of transactions, each made of 32 empty operations. Since there is only one thread and operations take no time, the results reflect the cost of concurrency management when there is no contention and no effective processing.

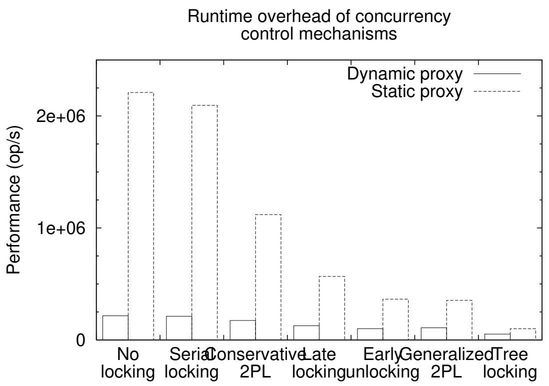


Figure 12: Runtime overhead of the different locking policies and interception mechanisms.

Results are shown in Figure 12. We have measured the cost of each locking policy with dynamic and static proxies. The column labeled "no locking" corresponds to execution of the application with the interception mechanisms but with no actual concurrency management. Serial locking acquires and releases a single global lock. 2PL locking policies acquire locks on all objects accessed by the transaction. Tree locking additionally locks and unlocks intermediary nodes of the tree.

The results are not surprising. Static proxies are clearly more efficient than dynamic proxies. The cost of using reflection to intercept invocations appears to be significantly bigger than the cost of concurrency management. In applications that perform time-consuming operations, the runtime overhead of dynamic proxies may be negligible in comparison to processing time. However, in applications that perform short operations, this overhead may become a bottleneck and static proxies should be preferred.

Among all locking policies, tree locking exhibits the highest overhead. This is easily explained by additional concurrency management performed on the nodes of the tree. Early unlocking and generalized 2PL pay the cost of post-invocation filters. Late locking performs slightly better because it only uses the less costly pre-invocation filters. Conservative 2PL does not use invocation filters at all and has the smallest overhead among 2PL policies. Finally, serial locking prevents concurrency by using a single global lock, thus minimizing runtime overhead. While these figures show the benefits of using simple locking policies, one has to balance the runtime costs with the increased concurrency of more complex locking policies. For application that perform time-consuming operations, concurrency must be the key factor for choosing a locking policy and runtime overhead should be ignored.

7. CONCLUSION

In this paper, we have presented mechanisms for implementing atomic sets of actions in Java. These mechanisms transparently manage isolation on a set of shared objects on behalf of the application, by increasing concurrency while preserving safety and liveness. They reduce the burden of the developer of concurrent applications, reduce the likelihood of semantic errors, and have the potential of increasing concurrency in complex applications.

We have presented various locking policies adapted to our application model, which consider a simplified form of transactions where operations are performed on transient data (no durability) and actions never need to be undone. Each strategy has specific benefits and drawbacks, and the choice of the best strategy ultimately depends on the nature of the application.

We have introduced several techniques used for the implementation of atomic blocks in Java and given some guidelines for choosing the technique best adapted to a given application. Transparent concurrency control management is achieved through object proxying. Finally, we have presented experimental results that illustrate the concurrency degree and runtime overhead of the various strategies discussed in this paper. These results show that there are tradeoffs between concurrency degree, runtime overhead, transparency, and flexibility.

We believe that basic mechanisms for atomic blocks would be a relevant addition to the Java language. A simple yet elegant approach for this purpose, without adopting all the features of our Java concurrency framework, consists in extending the "synchronized" keyword so that it can take an array of objects as argument and lock them conservatively using a deadlock-free conservative 2PL strategy.

8. REFERENCES

[1] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[2] J. Blosser. Explore the dynamic proxy api. *JavaWorld*, Nov. 2000. http://www.javaworld.com/javaworld/jw-11-2000/jw-1110-proxy.html.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[4] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[5] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, 1997.

[6] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[7] F. Schneider. *On Concurrent Programming*. Springer Verlag, 1997.

[8] P. D. Seymour and R. Thomas. Call routing and the ratcatcher. *Combinatorica*, 14(2):217–241, 1994.

[9] B. Shannon, M. Hapner, V. Matena, J. Davidson, E. Pelegri-Llopart, and L. Cable. *Java 2 Platform, Enterprise Edition: Platform and Component Specifications*. Addison-Wesley, 2000.

[10] A. Silberschatz and Z. Kedem. Consistency in hierarchical database systems. *Journal of the ACM*, 27(1):72–80, Jan. 1980.

[11] D. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, 1998.

[12] B. M. W.K. Edwards. *Core Jini*. Prentice Hall, 2000.

[13] G. Zipf. *Human Behaviour and Principle of Least Effort*. Addison-Wesley, Cambridge, Massachusetts, 1949.

APPENDIX

A. THE TREE-LOCKING PROTOCOL

The tree locking protocol follows these simple rules:

- A transaction T always starts by acquiring the lock on its root node, which is the lowest common ancestor of all the objects accessed by T.
- To access an object o, T follows the path that leads from the last accessed node (initially the root node) to the leaf holding o. On that path, T performs the following operations:
  - Let N be the current node in the path, and N' the next node in the path. T first acquires the lock on N' (if T does not already hold that lock).
  - If there is no object o' in the remaining operations of T such that N is an ancestor of o', then T releases the lock on N. (This situation happens if T has performed all its operations on the objects of a branch, and is moving upstream along the path.)
  - Otherwise, if for each object o' in the remaining operations of T such that N is an ancestor of o', N' is also an ancestor of o', then T releases the lock on N. (This situation happens if all remaining operations of T are confined in one branch, and T is moving downstream along the path towards that branch.)
- After its last operation, T releases the lock on the last accessed object.

Figure 13 shows an execution history of the transactions of Figure 4 with a tree locking policy. The tree is a balanced binary tree with three levels, three internal nodes, and four leaves. Transaction flow is represented by dashed arrows. Intermediary actions (i.e., locking, unlocking, operation execution) are indicated along the arrows as they occur. The different transactions on a figure execute concurrently and time flows in the direction of arrows.

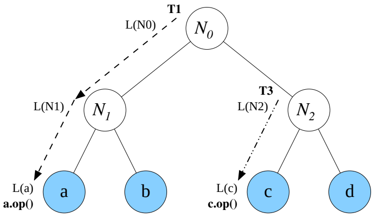
B. ADEQUACY CRITERIA FOR LOCKING TREES

We discuss below the four criteria that we have identified to characterize a good tree for a given set of transactions.

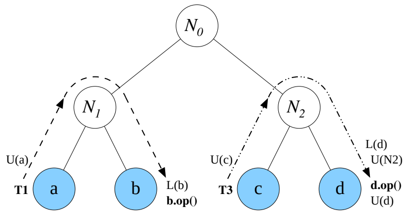
A first observation is that transactions can execute concurrently if they are confined in separate branches of the tree. Obviously, if all transactions compete to lock the root node of the tree, then concurrency will be equivalent to or worse than conservative 2PL. On the other side, if the root node of some transactions are in separate subtrees, they can execute in complete independence. *Criterion 1:* The lower the root node of a transaction is, the better concurrency is.

Nodes high in the tree are more crucial than nodes low in the tree, because they control a larger number of resources. It can be highly inefficient to lock a node high in the tree to access a single resource below that node. For instance, with the tree of Figure 13, a transaction that accesses a and c will prevent concurrent accesses to b and d because it locks nodes that control these objects. Concurrency is thus better if the transactions that lock a node access a large number of the resources controlled by that node. *Criterion 2:* The acquisition of a node must pay off and concurrency can be optimal when transactions access all resources located below that node.

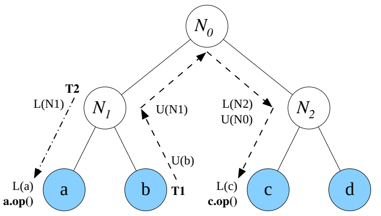
The order in which transactions access resources is also an important factor for concurrency. If a transaction leaves a subtree in which it will return later, it has to keep locks



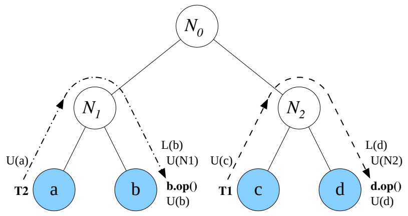
(a) T_1 executes in the left branch. T_3 can execute concurrently in the right branch.



(b) T_1 and T_3 do not interfere because they execute in separate branches. T_3 completes.



(c) T_1 moves to the right branch. T_2 can execute concurrently in the left branch.



(d) T_1 and T_2 finish their execution in separate branches without interfering.

Figure 13: Execution of the transactions of Figure 4 with a tree locking protocol.

on that subtree. On the other hand, if a transaction leaves a subtree definitively, it can release the locks it holds on the subtree. Therefore, if all accesses to the resources of a subtree are adjacent, the transaction can release all locks on the subtree when leaving it. This is the case of T_1 with the tree of Figure 13: once T_1 has accessed a and b, it can leave the left branch of the tree and release all the locks it holds on that branch (Figure 13(c)). *Criterion 3:* Concurrency is increased if accesses to the resources of a subtree are adjacent in a transaction.

The first three criteria apply to individual transactions, i.e., they define if a tree is adequate for each transaction in isolation. A fourth criterion can be defined on sets of transactions. It derives from the observation that, if multiple transactions access the same subtrees, concurrency can be increased if they access these subtrees in the same order. For instance, in the tree of Figure 13, if we define a new transaction T_1' which accesses the same objects than T_1 in the same order, T_1' can start executing in the left branch as soon as T_1 moves to the right branch. If T_1' was accessing objects in the reverse order than T_1, then it would have to wait until T_1 completes before starting execution. *Criterion 4:* Concurrency is generally increased if shared resources are accessed by multiple transactions in the same order.

C. THE OPTIMAL TREE PROBLEM

The Problem 4.1 can be proved to be NP-hard. While the details of the proof are outside the scope of this paper, the intuition behind the proof can be outlined by considering the special case in which each of $T_1, ..., T_n$ is of size two, and accesses two distinct objects. The problem of finding an optimal tree for these transactions is easily seen to be equivalent to the following problem: given a graph $G = (V, E)$ and a weight function $w : E \to N$, construct a routing tree T for G, i.e., a tree T in which each internal node has degree 3 and the leaves correspond to vertices of G, such that the congestion at each internal node of T is minimized. The congestion at a node of the routing tree is the maximum, for any vertex x, of $\sum_{(u,v)\in E, u\in S, v\notin S} w(u, v)$, where S is one of the three connected components obtained by deleting x from T. The tree that minimizes B can be shown to be equal to F_{min} and F_{avg} with the algorithm of Section 4.2, when the weight $w(u, v)$ of each edge (u, v) of G corresponds to the number of occurrences of the transactions $\{u; v\}$ or $\{v; u\}$. Seymour and Thomas proved in [8] that a closely-related problem, where the congestion of the routing tree must be minimized at its edges instead of its nodes, is NP-hard. Our problem can also be shown to be NP-hard by extension of Seymour and Thomas' results.

As a result, we have primarily focused on heuristics for building a *good* tree in polynomial time. Since the nodes of a tree are artificial objects that are not associated with data, their number and structure is very variable. Although we presented a balanced binary tree in Figure 13, the algorithm of Section 4.2 does not impose restrictions on the *depth* of a tree or the *arity* of any of its nodes.[5] These factors influence the performance of the tree locking protocol and must be chosen accordingly.

[5]In fact, with our simplified model, for any tree with some nodes of arity greater than 2 it is possible to find an equivalent binary tree, at the price of increased depth.

D. TREE CONSTRUCTION ALGORITHM

Given a set of n transactions $T_1, ..., T_n$ with sizes $m_1, ..., m_n$, where each transaction T_i is composed of m_i individual operations $o^i_1, ..., o^i_{m_i}$ on shared resources $r_1, ..., r_l$. Informally, our greedy algorithm for building binary locking trees works as follows:

1. Shared resources $r_1, ..., r_l$ are organized in an "access" graph $G = (V, E)$ with a weight function $w : E \to N$ such that $w(r_i, r_j)$ is a pair of values (w_d, w_n): w_n is the number of occurrences of operations on both r_i and r_j in each transaction $T_1, ..., T_n$, and w_d is the sum of the distance between these operations. Let L be an (ordered) list initially empty.
2. Select the vertex u that maximizes $\sum_{(u,v)\in E, v\in V} w_n(u, v)$. If there is more than one candidate vertex, select the one that minimizes $\sum_{(u,v)\in E, v\in V} w_d(u, v)$. Add u to L.
3. Select the vertex $u' \notin L$ that maximizes $\sum_{(u',v)\in E, v\in L} w_n(u, v)$. If there is more than one candidate vertex, select the one that minimizes $\sum_{(u',v)\in E, v\in L} w_d(u', v)$. Append u' to the end of L. Repeat this step until L contains all vertices of V.
4. Create a balanced binary tree with l leaves and arrange the resources the resources $r_1, ..., r_l$ in the leaves of the tree in the same order as they appear in L.

For instance, given the transactions T_1, T_2, and T_3 in Figure 14, the algorithm will produce the graph in Figure 15 and the same tree as in Figure 6.

$$\begin{array}{lcl} T_1 & : & a.op()\ ;\ b.op()\ ;\ a.op()\ ;\ c.op() \\ T_2 & : & c.op()\ ;\ b.op()\ ;\ d.op() \\ T_3 & : & a.op()\ ;\ b.op() \end{array}$$

Figure 14: Three sample transactions.

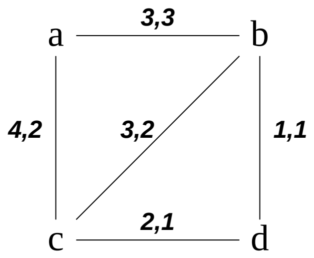


Figure 15: Access graph for the transactions of Figure 14.