High-Performance, Space-efficient, Automated Object Locking

Laurent Daynès Grzegorz Czajkowski
Sun Microsystems Laboratories
901 San Antonio Road, Palo Alto CA 94303
{firstname.lastname}@eng.sun.com

Abstract

The paper studies the impact of several lock manager designs on the overhead imposed to a persistent programming language by automated object locking. Our study reveals that a lock management method based on lock state sharing *outperforms more traditional lock management designs. Lock state sharing is a novel lock management method that represents all lock data structures with equal values with a single shared data structure. Sharing the value of locks has numerous benefits: (i) it makes the space consumed by the lock manager small and independent of the number of locks acquired by transactions, (ii) it eliminates the need for expensive book-keeping of locks by transactions, and (iii) it enables the use of memoization techniques for whole locking operations. These advantages add up to make the release of locks practically free, and the processing of over 99% of lock requests between 8 to 14 RISC instructions.*

1. Introduction

The PJama Virtual Machine (PJVM) [1] is an extension of the Java^{TM} Virtual Machine (JVM) [2] with orthogonal persistence [3]. From the outset, the PJama project has aimed at a flexible and integrated transaction (FIT) environment, where every computation runs in the context of a transaction, and all objects, irrespective of their type and lifetime are under transactional control [4]. The FIT combination of type safety, systematic confinement of computation in transactions, and automatic enforcement of transactional properties, offers a robust environment that makes it unnecessary to use separate virtual machines (and therefore, separate operating system processes) to defend persistent applications against each other's errors.

One of the main obstacles to applying the FIT approach to PJama is the absence of sound technology to efficiently support automated concurrency control. We favored locking over other concurrency control techniques for two main reasons: its impact on PJama's already sophisticated memory management is minimal; and, it can be easily extended to support advanced concurrency control semantics [4], which will be supported in future versions of PJama.

PJama differs from most multi-user persistent programming languages (PPLs) and object-oriented databases (ODBMS) in that all transactions execute within the same address space, and can directly access both transient and memory-resident persistent objects. This architecture requires sophisticated lock management, such as that used in centralized relational database systems, to mediate the direct accesses to objects by transactions.

1.1. Problem Characterization

Programming systems that thightly couple a transaction processing engine with a high-level programming language usually include in their runtime a component that automates the requesting of locks on behalf of executing programs. Automated locking both simplifies the programmer's work and avoids depending on the programmer to always formulate *well-formed* transactions, that is, transactions that execute an operation on an object only when they own the lock of that object in the lock mode corresponding to that operation. In a PPL, automated locking is accomplished by transparently augmenting programs with small sequences of instruction, called a *lock barrier*s, that issue lock request to the lock manager (LM) when it is appropriate to do so.

Ideally, a transaction needs to request the lock of an object it accesses only once, before its first access to that object. Identifying ahead of time the first access to an object by an arbitrary program is, in general, impossible. A pragmatic solution is to precede every object access with a lock barrier, and rely on compiler analysis to identify and remove as many *redundant* lock barriers as possible.

Implementing automated locking at the granularity of individual objects in PJama is challenging. First, the size of objects is small, typically between 16 to 42 bytes [5], while transactions can be very large. For instance, the OO7 benchmark [6], which is believed to reproduce the behavior of a typical PPL application, defines elementary operations

that access 60,000 objects for small-size databases. This requires space-efficient lock management that scales well with the number of locks. Second, the characteristics of the Java^{TM} programming language complicate the elimination of redundant lock barriers. Dynamic class loading prevents the use of global optimization, the costs of which cannot be afforded at runtime anyway. The absence of effective methods for removing unnecessary lock barriers results in frequent unnecessary lock requests (99% as indicated by our measurements). Lastly, persistent objects typically reside for a substantial amount of time in main memory so that the use of *swizzling* techniques [7] to translate them into a main-memory format suitable for direct manipulation pays off. Thus, accessing a persistent object often costs as little as a main-memory access. Hence, each instruction added by a lock barrier to an object access has dramatic performance consequences.

1.2. Related Work

We aren't aware of any PPLs or ODBMSs that use an architecture similar to that of PJama, which combines direct access by concurrent transactions to a shared heap of objects with automated fine-granularity locking. PPLs that offer transactions (e.g., [8, 9]) leave the responsibility for concurrency control to the programmers, who have to manually set locks. Most ODBMSs have adopted a client-server architecture, where clients supply each transaction with a private buffer of objects [10, 11, 12, 13], or a private buffer of pages [14]. Some ODBMSs (e.g., [15]) combines direct access by multiple transactions to a shared page buffer pool with page locking in order to use virtual memory protection to automate the acquisition of locks. This approach eliminates the need for efficient software-only read and write lock barriers. However, it constrains the granularity of locks to be a multiple of the page size, it requires a separate address space per transaction, and it prevents objects from moving from their original virtual page.

Enabling direct access by concurrent transactions to a shared heap requires a sophisticated lock manager, such as those used in centralized relational database systems. The implementation techniques of lock managers (LMs) have not evolved significantly during the last twenty years [16], and all database systems seem to use some minor variation of the System R lock manager [17]. The most complete and recent detailed description of a LM can be found in [18]. In short, each lock is represented by a data structure called a lock control block (LCB). An LCB is the head of a list of lock request control blocks (LRB). Each LRB represents a granted or pending request of one transaction for the lock. Transactions keep track of their locks by chaining all their LRBs together. LCBs are maintained in a hash table keyed by resource identifier. Processing a lock request consists of looking up the lock hash table for an existing LCB, and allocating one if none was found. Then, the LRB chain is searched to find the LRB of the requester. If none is found, an LRB is created for the requester to represent either a pending or a granted request, depending on what the conflict detection diagnosed.

Rather than revisiting this design, database implementors have focused on reducing the number of calls to the LM and the overall number of locks used by taking advantage of the physical organization of data into container hierarchies, the limited number of access paths to the data, the semantics of operations on data, and the query-oriented nature of accesses to the data (range-locking techniques [19, 20] illustrate this well).

The introduction of main-memory database systems has changed the trade-off between tuple access and lock management costs, making the database community look closer at the performance of lock operations. The main novelty is the replacement of the hash table that maps locks to the resource they protect, by direct pointers from resources (e.g., tuple) to locks [16], since locked resources are always memory resident. To reduce the high cost both in terms of space and processing overhead of record locking, lock de-escalation techniques [16] and cheap-first locks [21] were suggested. The former relies on the premises of a query processing programming interface, built-in indexes, and a small fixed number of paths to data, and is therefore inappropriate for PPLs. We used a variant of the later in one of the implementations studied in this paper.

1.3. The Focus of the Paper

Our previous work revisited the underlying principle of lock management to eliminate the components of a LM that do not scale with the number of locking units. The result was a novel approach which we called *lock state sharing* [22]. Lock state sharing was shown to dramatically reduce the space overhead of object locking.

This paper focuses on the processing overhead of automated locking. Our previous work was rather inconclusive with respect to this dimension, mostly because of the poor performance of the version of the PJVM used then – an interpreter-only JVM based on the JDK^{TM} version 1.1.7. The PJVM we used this time is based on a state-of-the-art, handle-less, JVM, that includes better garbage collection support, faster synchronization, native thread implementation, and a well-tuned just-in-time (JIT) compiler. It runs 10 times faster than the previous one [24], and adds about 20% of overhead to programs when compared with the original JVM it is derived from[1]. This dramatic improvement to the

[1]The execution times for OO7 's traversals executed on a transient database is 21% slower than with a non-persistent JVM. The SPECjvm98 *db* and *javac* programs are slowed down by 14% and 18% respectively.

performance of the PJVM makes it much more sensitive to the choice of a particular combination of lock barrier and LM to implement automated locking.

The paper is organized as follows. Sections 2 to 5 briefly describe the LM designs studied in the paper. Section 6 presents how the PJVM was modified to support automated locking. Section 7 describes the experiments we conducted to evaluate the impact of the various lock management techniques on PJama's performance, and Section 8 analyzes their results. Section 9 reports our conclusions.

2. "Traditional" Lock Management

The LM used as the starting point of this study derives from our previous work on PPL-friendly LMs [25], because it was shown to outperform more traditional implementations (e.g., [18]). However, this LM still shares many of the general principles of all the LMs we are aware of, so we will refer to it as *TRAD*, for "*traditional*" implementation. Since all the other LMs studied in the paper result from modifications to *TRAD*, we review its most salient features below.

2.1. Fast access to locks

Fast access to the lock of an object is obtained by storing in its main-memory representation a direct pointer to its lock. In constrast to memory-resident database systems [16], we also face the problem of maintaining the object-to-lock association across possible movements of the object between main memory and disk.

The PJVM, like most persistent object system implementations, maintains in main-memory an associative table, called the ROT (Resident Object Table), that maps persistent object identifiers to addresses in main memory [10, 11, 24]. When the memory manager evicts an object, it first determines whether the object is locked, and if so, replace its address in main-memory with that of its lock. The lock pointer thus remembered is re-installed in the object it protects the next time this one is faulted-in. ROT entries of unlocked non-resident objects are recycled.

2.2. Compact static representation of locks

Figure 1 shows the overall design of the *TRAD* LM. A fixed-size bitmap representation of locks was favored over the classic linked list of lock LRBs. Each active transaction is given a *locking context*, which the LM identifies by a unique bit number. The bit number also indexes a table of locking contexts. All bitmaps use the same mapping from locking contexts to bit numbers, i.e., the ith bit always identifies the same locking context, and therefore, the same transaction. A bit set to 1 in a bitmap indicates that the transaction corresponding to this bit location belongs to the set of transactions represented by that bitmap.

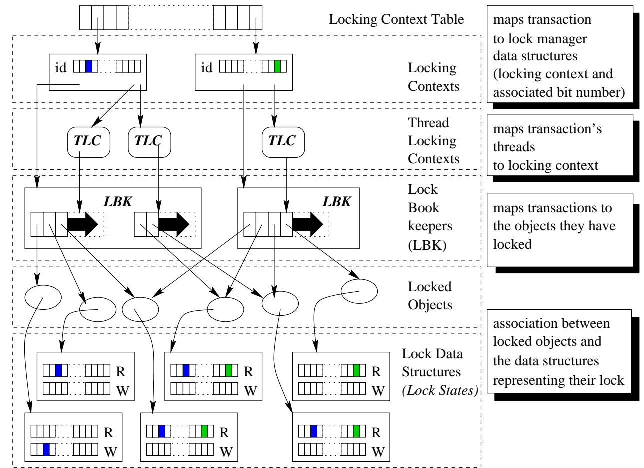

Figure 1. The *TRAD* lock manager design.

The locking context of a transaction comprises its bit number, a list of *thread locking contexts*, and a lock book-keeper. Thread locking contexts are a thread-private data structure used by the LM to service a thread's lock requests with minimal synchronization between threads of the same transaction. Similarly, lock book-keepers maintain a pool of thread-private growable stacks of references to locked objects.

Locks are implemented as *lock state*s. A lock state consists of a *lock type*, book-keeping data, and an array of bitmaps. Each bitmap represents an *owner set* for a given lock mode. A transaction T has been granted a lock L in a mode M if T belongs to the owner sets of L associated with M.

This design has several advantages: (i) lock data structures being of fixed size, at most one transaction pays a memory management operation per lock; (ii) the space overhead is quickly amortized as the working sets of concurrent transactions overlap, whereas, with an LRB list, it increases with the number of transactions [25]; and (iii), operations such as testing and updating lock ownerships translate into fast logical operations on bitmaps. The size of bitmaps is chosen to match the maximum number of concurrent transactions supported.

2.3. Cheap-First Lock

In order to reduce both the space and processing overhead of locking, the allocation of a lock structure is postponed until a second transaction requests a lock. This idea, referred to as "cheap-first lock", is credited to IBM's IMS Fast Path [16, 21]. Cheap-first locking marks objects locked by

a single transaction with a tag that identifies both the lock's owner and mode. A lock data structure is allocated only when a second transaction requests a lock on that object.

The *TRAD* LM implements cheap-first locks as follows. All unlocked memory-resident objects point to an *immutable* lock data structure set with the unlocked value, and shared among all unlocked objects. The locking context of each transaction is augmented with two immutable lock data structures that represent the value of a lock owned by that transaction only in, respectively, read and write mode. These lock values are called, respectively, the SRO (single read owner) and SWO (single write owner) of the transaction. The pointers to the SRO and SWO are the tags used to mark objects locked by that transaction only. Thus, the lock pointer of all objects locked by a single transaction in a given mode points to the same lock data structure.

When a transaction requests a lock against an unlocked object, it atomically exchanges the lock pointer to the unlocked lock state with the lock pointer to either its SRO or SWO, depending on the locking mode requested, and adds the reference of the object to its lock book-keeper. When a transaction requests a lock on an object already associated with either an SRO or an SWO lock value for another transaction, it allocates a new lock data structure, sets its new value, and atomically exchanges the current lock pointer with that of the allocated lock structure before recording the locked object. Inversely, when a lock-release operation results in a single-owner lock value, the lock pointer of the locked object can be replaced with the corresponding SRO or SWO pointer, so that the space of the previous lock data structure can be reclaimed.

2.4. Fast-paths

Automated locking can generate an overwhelming number of redundant lock requests, i.e., requests for locks already granted. Because rigorous 2PL is enforced, once granted, a lock is not released until a transaction that owns the lock completes. To reduce the overhead of redundant lock requests, a transaction can take an inconsistent view of the state of a lock and test if it owns it. Such inconsistent tests can be small enough to be inlined in the LM's caller. In the following, we call such small sequences of inlined code a *fast-path*, for it can bypass a call to the LM.

The simplest fast-path is a test of the lock pointer of the requested object. If it is equal to one of the cheap-first lock values of the requester, the LM is bypassed. This fast-path takes only 5 instructions, but can fail often because it covers only a few cases of redundant lock requests.

A more general fast-path consists of testing the membership of the requester in the owner set corresponding to the requested lock mode. This fast-path takes between 8 to 10 instructions, depending on whether the size of bitmaps implementing owner sets is larger than the size of a register.

3. Lock State Sharing

The *TRAD* implementation suffers two major drawbacks that are the consequence of the underlying principles that are used in all of the LMs of which we are aware: (i) *there is one lock data structure per unit of locking*, and (ii), *the lock manager keeps track of the locks of each transaction in order to automatically release them when a transaction terminates*. Delaying lock data structure allocations using cheap-first locks only helps when transactions rarely overlap their working sets.

The *lock state sharing* (*LSS*) technique introduced in [22] to correct the first problem relies on two observations: the number of running transactions is very small when compared to the number of objects they access; and the number of combinations of sharing between transactions is likely to be small. In other words, one can expect many lock data structures to have identical values.

Therefore, it is advantageous to make two objects with locks of *equal value* point to the same *shared immutable* lock state, and have lock operations change the *association* between an object and the shared lock state representing the value of its lock, rather that updating a private lock data structure. To guarantee that only one shared lock state is used to represent a given lock value, the LM maintains an associative table of immutable shared lock states (*TILS*) keyed on lock values. Note that the TILS does not hold all possible lock values: it is initially empty, and shared lock states are added to it as needed, when a TILS lookup fails. Garbage collection techniques determine unused shared lock states and remove them from the TILS.

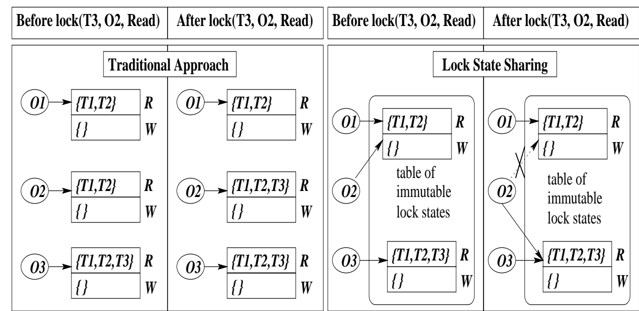

Figure 2. Locking with *TRAD* vs *LSS*.

Figure 2 compares *LSS* (right) with the traditional approach (left) using a simple example. It shows a scenario where two objects, O_1 and O_2, have been locked in read mode by two transactions T_1 and T_2, and a third object O_3 has been locked in read mode by transactions T_1, T_2 and T_3. The state of each lock and the association between objects and

their lock is shown before and after the acquisition of a read lock on O_2 by T_3.

In the traditional approach, each object is associated with a private lock data structure. T_3's request for O_2's lock is processed by updating O_2's private lock data structures to reflect the lock's new value (i.e., lock owned in read mode by T_1, T_2, and T_3). In *LSS*, O_1 and O_2 share the same lock state which holds the value that the two private data structures representing their respective locks would have. Instead of updating the lock state data structure associated with O_2 to process T_3's request, the LM searches the TILS for a shared immutable lock state with O_2's new lock value, and atomically exchanges O_2's current lock pointer with the pointer returned by the TILS. It is crucial to understand that in both approaches, every object is a unit of locking: the sharing of lock states of equal value must not be confused with protecting several objects using the same lock.

LSS is obtained with minimum changes to the *TRAD* implementation: lock data structures remain unchanged, except that they now represent shared immutable lock states instead of mutable private locks. A hash table of lock data structures keyed on their value implements the TILS. Locking operations work by first building a temporary lock-state value equal to the new value the object's lock must have, and then by using it to probe the TILS for an equivalent immutable shared lock state. The object's current lock pointer is then atomically exchanged with the pointer to the shared lock state returned by the TILS.

To summarize, *LSS* allows transactions to arbitrarily overlap their working sets without dramatic space consumption due to locking. *LSS* is much less sensitive to the maximum size of owner sets, and can afford the use of owner sets containing several hundred elements without noticeable space overhead. Lastly, transitions from one lock state value to another one consist of a single pointer exchange. On most stock hardware this can be accomplished with a single atomic compare-and-swap instruction, such as the SPARC^{TM} platform V9 `cas` or Intel486's `cmpxchg`. This promotes the use of non-blocking synchronizations to implement locking operations, which avoids expensive latching and reduces contention.

4. Elimination of Lock Book-Keeping

All the LMs presented so far keep track of the objects locked by each transaction in order to automatically unlock them when the transaction completes. This stems from the original database systems' assumptions that (i) the total number of lock data structures is much larger than the number of locks acquired by one transaction, and (ii), transactions acquire a relatively small number of locks[2]. Under these conditions, keeping track of a transaction's locks is efficient.

[2]Database LMs have recourse to adjustable locking granularity to keep the number of locks below a few thousand [18].

LSS invalidates the above assumptions. Indeed, the total number of shared immutable lock states maintained by the LMs is expected to be several orders of magnitude smaller than the number of objects locked by a single transaction. Hence, to release the locks of a transaction, it is beneficial to simply scan all the shared lock states to find those that represent locks of that transaction, and update their values to reflect the effect of a lock release. This relaxing of the immutability of shared lock states raises two issues. First, it might generate duplicates of existing shared lock states. Such duplicates do not harm the correctness of the locking logic, but increase space consumption. Second, synchronizing exchanges of pointers to shared lock states with updates to shared lock states can increase substantially the costs of lock acquisitions.

The *NLSS* LM is derived from the *LSS* LM by eliminating lock book-keeping data structures and related code. When a transaction T releases all of its locks, the *NLSS* LM first scans the TILS to find all the shared lock states that represent values of locks held by T. For each such lock state, T is removed from all the owner sets. This might turn the shared-lock state into a duplicate of another existing shared lock state of the TILS. If that is the case, the modified lock state is removed from the TILS and added to a *list of duplicates*. Otherwise, it is re-hashed into the TILS.

Once the TILS is processed, the LM must take care of duplicates that might represent values of T's locks. Such duplicates may have been created by transactions that were running concurrently with T but completed before T, and whose working sets overlapped with that of T. The LM scans the list of duplicates and updates those that hold lock values representing T's locks. Modifying a duplicate of a TILS's shared lock state always results in a duplicate of another TILS's shared lock state. Therefore, a duplicate can never re-enter the TILS.

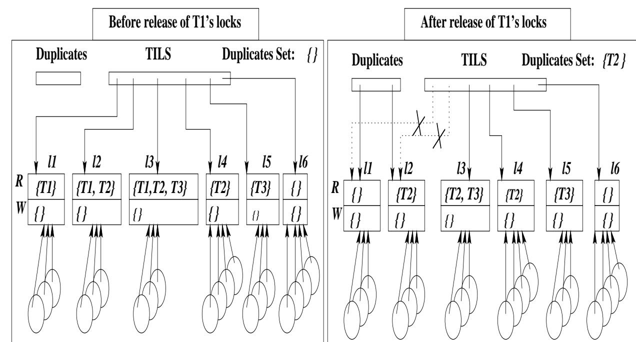

Figure 3. Releasing of locks with *NLSS*.

Figure 3 illustrates how *NLSS* works. It depicts the data

structure of the LM before and after the release of the locks of transaction T_1. Before T_1's end, three shared lock states represent values of lock owned by T_1 (l_1, l_2 and l_3). When T_1 completes, it is removed from the owner sets of these lock states. However, the new updated values of two of these lock states (namely, l_1 and l_2) are already represented in the TILS (by, respectively, l_6 and l_4), so they are removed from the TILS and added to the list of duplicates. When T_2 terminates and releases its locks, it will generate two additional duplicates of the "unlocked" lock state l_6 (l_2 and l_4) and a duplicate of l_5 (l_3). Note that *no associations* between locked objects (circles in Figure 3) and shared lock states are modified when a transaction releases its locks.

The elimination of lock book-keeping from *LSS* seems attractive because it saves both computational and space costs. Furthermore, the costs of releasing all of the locks of a transaction is in terms of the number of shared lock states instead of the number of objects locked by that transaction. On the other hand, the absence of lock book-keepers generates duplicates that increase space consumption, and requires potentially costly sophisticated synchronizations.

5. Elimination of TILS Lookups

The most expensive part of a locking operation for the *NLSS* lock manager is the probing of the TILS to obtain the shared lock state that represents the new value that the object's lock must have. If that shared lock state were already known, the locking operation would just consist of an atomic exchange of the object's current lock pointer with that lock state. Knowing the lock state that represents the result of a lock request in advance can be achieved by applying *memoization* - a technique used in functional programming languages [23].

More specifically, let T be a transaction, *op* a locking operation, l_i and l_f two shared lock states such that l_f holds the lock state value equal to the result of T executing *op* with the lock value held by l_i (i.e., $l_f = op(T, l_i)$). Instead of computing the value of $op(T, l_i)$ upon every call to *op* and probing the TILS with that value to obtain the equivalent shared lock state l_f, the LM maintains l_f in a cache keyed by *op*, l_i and T. If the parameters passed to a call to *op* match l_i and T, then l_i can be immediately exchanged with l_f. Otherwise, $op(T, l_i)$ is computed, the TILS probed, and the cache updated before atomically changing the shared lock state pointers.

Adding memoization to *NLSS* is straightforward: each thread locking context is augmented with a memoization cache. Each cache line consists of two shared lock state pointers. The cache has a *single line* per locking operation (e.g., read lock acquisition, write lock acquisition, lock release), that is, only one result per operation is memoized. Probing the cache and subsequently exchanging the current lock pointer with the probe's result upon a cache hit takes 6 instructions (one arithmetic computation of the lock pointer address, two loads, one atomic compare and swap, a comparison and a branch). This variant of *NLSS* is called m-*NLSS*.

6. Automated Object Locking

Both the interpreter and the JIT compiler of the PJVM have been modified to precede every object access with a lock barrier. Lock barriers typically consist of a call to the LM wrapped in an inlined fastpath (if one is defined by the particular lock manager used). Some of the optimizations already performed by the PJVM JIT compiler, such as common sub-expression elimination, naturally eliminates some redundant lock barriers, but only in marginal proportion, and with almost no visible effect on performance.

One negative effect of this strategy is to pair most accesses to *new objects*, i.e., objects allocated by transactions in progress, with a lock barrier, although locking new objects is unnecessary to enforce strict isolation. New objects fall into two categories: objects reachable only from the stack of the thread that created them, and objects whose reference has been stored by their creator into objects reachable from other transactions. Objects of the first kind are unreachable from other transactions. All paths to objects of the second category are protected by an exclusive lock of their creator, since in order to store a reference in an object O, the automated locking system first obtains a write lock for O. In both cases, access to new objects by transactions different from their creator is already prohibited, therefore locking is unnecessary for new objects.

One simple strategy to reduce the overhead of lock barriers against new objects is to make them look as if they are already locked by their creator. Under this approach, new objects are associated with the SWO of their creator at allocation-time, without calling the LM. Upon transaction completion, the shared SWO lock value is atomically turned into a shared unlocked lock value. All of the created objects look like they are unlocked and become subject to normal automated locking rules. This strategy turns all requests to new object's locks into redundant lock requests, and guarantees that new objects escape to lock book-keeping.

7. Performance Experiments

A total of 15 different PJVM versions were evaluated. The versions differ from one another by two parameters: the LM and the lock barrier used. Table 1 lists the various versions and their characteristics. The LMs share most of their code, except for lock operations whose implementation is specific to each LM. The number of concurrent transactions

was limited to 64 for all LMs so that transaction set operations could be implemented with single 64-bit register instructions. This case favors *TRAD* because both the processing of bitmap operations and the space consumed for a lock state is minimal.

lock barrier			
direct LM call	single owner	single write owner	owner set membership
TRAD	so-*TRAD*	swo-*TRAD*	osm-*TRAD*
LSS	so-*LSS*	swo-*LSS*	osm-*LSS*
NLSS	so-*NLSS*	swo-*NLSS*	osm-*NLSS*
m-*NLSS*		swo-m-*NLSS*	osm-m-*NLSS*

Table 1. Lock Barrier Implementations.

The performance experiments were done using SPECjvm98 [26], and OO7 [6]. SPECjvm98 is a standard suite of benchmark programs destined to evaluate the performance of JVMs. Due to space limitations, we only report measurements for db (multiple database functions on a memory-resident database), and javac (the Java compiler from the JDK^{TM} version 1.0.2).

The OO7 benchmark synthesizes applications managing complex data structures such as CASE or CAD/CAM systems. It defines a database organized in modules. Each module has a manual and a seven level deep hierarchy of assembly objects. Assemblies recursively reference three other assemblies. Leaf assemblies have three bi-directional links toward three composite parts, each of them consisting of a graph of atomic parts inter-connected to three other atomic parts of the same graph. Inter-connections are themselves objects. The benchmark defines three database configurations which varies the number of atomic parts per composite part, and the size of the database. Our experiments with OO7 focus on read-only traversal operations, namely T1 and T6. as we study the overhead of locking operations, and not throughput. Using read-only operations allows for experimenting with various degrees of working set overlap between transactions without interference due to conflicts. Traversal operations navigate through the assembly hierarchy and visit each composite part of each base assembly. For each visited part, T1 performs a depth-first traversal on its graph of atomic parts, whereas T6 visits the root of the graph only.

Measurements were done on a Sun Enterprise^{TM} 420, with 4 UltraSPARC^{TM}-IIi processors clocked at 450Mhz, a system clock frequency of 113Mhz, 1 Gb of main memory, running the Solaris^{TM} 2.7 operating environment.

8. Performance Analysis

All measurements are compared with measurements of the original PJVM, which does not support transactions. The data reported includes the count of lock requests issued for each benchmark in order to evaluate the fraction of requests that result in calls to the LM. In all our measurements, calls originating from the interpreter accounted for less than 0.05%, so we will not discuss them.

8.1. SPECjvm98

The left part of figure 4 reports the performance of two of the SPECjvm98 programs (the others behave similarly.) These programs were run unchanged. In that case, the static `main` method is wrapped by a *main* transaction automatically started by the transaction manager, and the whole computation takes place in that transaction. All the objects manipulated by SPECjvm98 programs are created by the main transaction only. As seen in Section 6, the solution to compensate for the lack of program analysis to identify new objects is to make them look as if they have already been locked. Consequently, all lock requests issued for these objects by the automated locking system are redundant. Performance then depends on how well the lock barrier used can filter unnecessary lock requests.

Two histograms are shown for each program: the rightmost one shows the overhead relative to a PJVM without automated locking, the one next to it shows the number of lock barriers that resulted in a call to the LM, compared with the total number of lock barriers executed.

The three lock managers *TRAD*, *LSS*, *NLSS* process lock requests for a lock owned *only* by the requester in exactly the same way. Hence there is little difference between systems in which lock barriers call directly the LM(*TRAD*, *LSS*, *NLSS*). Although owner-set membership (OSM) tests take more instructions than a single-owner (SO) tests, they filter *all* redundant lock requests. SO tests often fail because the majority of lock requests are read lock request for new objects protected by write locks. Hence the heuristic “test lock pointer against the requester’s SRO lock pointer” use for read lock requests systematically fails. As a result, automated locking systems that use SO tests perform worse than their counterparts that call the LM directly, because the extra instructions of the test never pay off. As an experiment, we changed all the SO tests so that the current lock pointer is systematically tested against single write-ownership (SWO) by the requester (which covers both read and write lock ownership). This time, most lock barriers succeed and the locking overhead is about the same as with the OSM tests.

Memoization alone also fails to filter all of the calls to the LM, because of the mix of lock request which decreases the

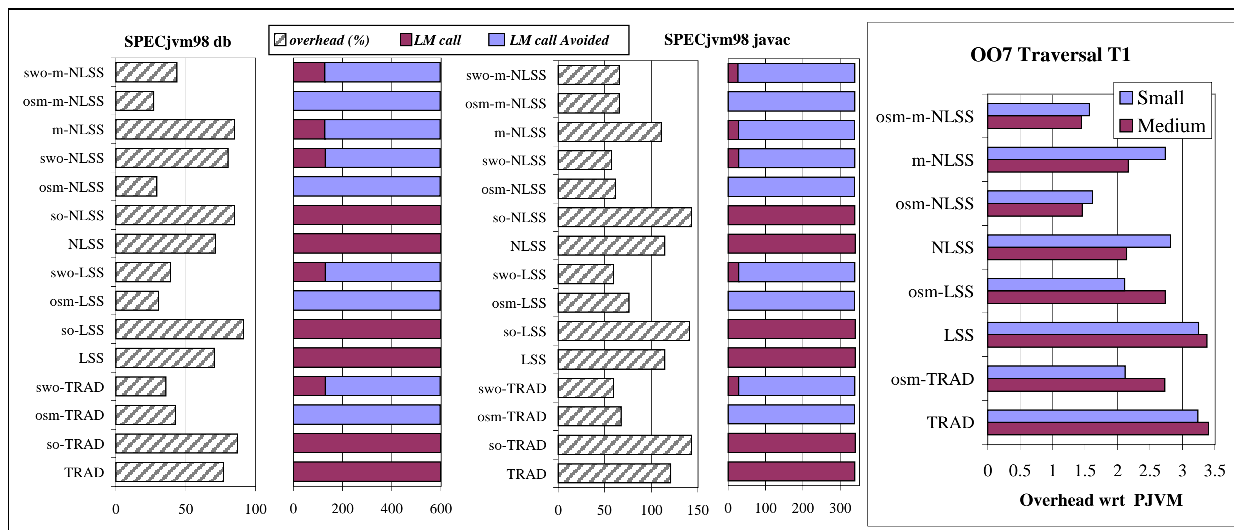

Figure 4. Overhead with respect to PJVM: left, SPECjvm98 programs db and javac, right OO7 traversal T1, small and medium database.

efficiency of the memoization cache. In this case, preceding the use of the memoization cache with an OSM or a SWO test also pays off.

Overall, for non-transactional workloads, automated locking imposes a minimum overhead of between 26% (SPECjvm98 db) and 70% (SPECjvm98 jack, a parser generator).

8.2. OO7

For the OO7 benchmark the individual traversal operations are run as transactions such that the measured transaction entirely overlaps its working set with a number of other transactions. This 100% overlap among transaction working sets defines the upper bounds for the time and space overheads of locking. To measure this, the first $n-1$ transactions are started, invoked to run the traversal once, and suspended just before commit. The nth transaction is then executed to completion and measured. Figure 5 reports the measurements of that n^{th} transaction for various degrees of overlap ($n = 0$ if there is no overlap) for two of the read-only traversals of OO7, namely T1 (on the left) and T6 (middle), executed against the **medium** size database. We report only measurements of versions that bring additional insight about the costs of automated object locking with respect to Section 8.1. Therefore, we will not discuss versions that use lock barriers based on SO tests (they consistently underperform direct call to the LM), and lock barriers based on SWO tests (they perform up to 34% worse than OSM tests).

The right part of figure 4 compare the overhead with respect to the PJVM of a traversal T1 of the small and medium databases. The histograms on the rightmost part of Figure 5 show how many calls to the LM are avoided by each of the four type of lock barrier we experimented with (namely, OSM test combined with memoization, memoization alone, OSM test alone and direct call to the LM). The bars reporting “filtered” reads and writes show the amount of read and write lock requests that were successfully processed by the lock barrier without entering the LM.

In contrast to the non-transactional workloads of SPECjvm98, the choice of a lock management method now matters because of the number of lock acquisition and release operations: a traversal T1 of a medium database acquires about 625,000 locks.

In the absence of overlapping transactions, redundant lock requests are still the dominant source of locking overhead (they account for 99.2% of the lock requests issued.) Therefore, the use of OSM tests reduce the locking overhead substantially. Lock acquisition and release are fairly cheap in the absence of overlapping transactions since, for all LMs, they merely consist of an atomic exchange of pointers to immutable shared lock state (lock acquisition swaps pointers to the unlocked shared lock state to the requester’s SRO, and the other way around for lock release). In particular, neither *LSS* not its variants need to probe the TILS to acquire lock. Avoiding lock book-keeping pays off, as illustrated by the gap between osm-*TRAD* and osm-*LSS* on one hand, and osm-*NLSS* and osm-m-*NLSS* on the other hand.

When a transaction overlaps with another one, locking

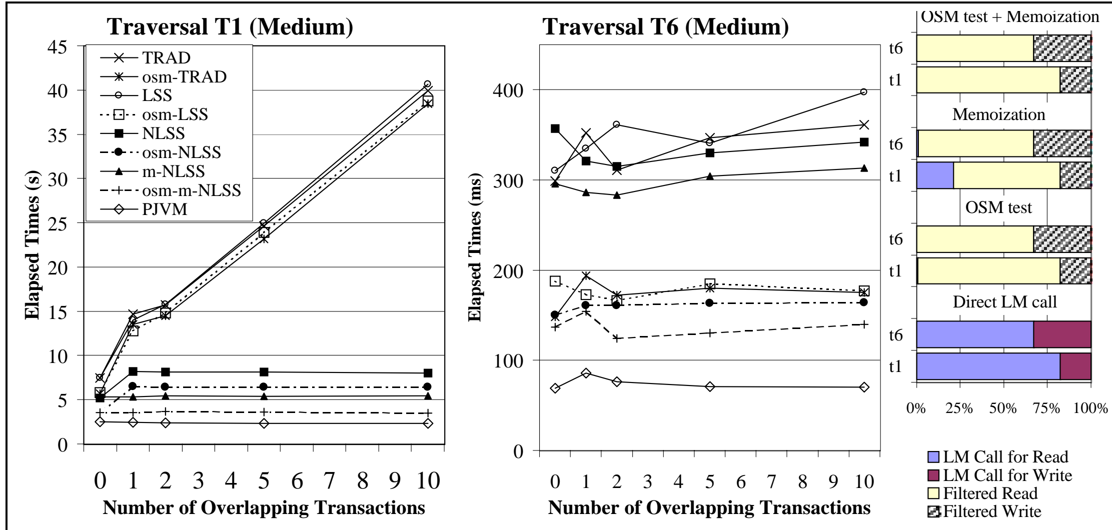

Figure 5. Measurements for OO7 Traversals (100% of overlap between transactions).

operations become more expensive. In particular, for *TRAD* and its variants, the second transaction pays the price of dynamic memory management to allocate/free private lock data structures for each traversed object, according to the cheap-first lock strategy. In constrast, lock state sharing implementations allocate only one new lock data structure to represent the new lock value. One the other hand, the costs of probing the TILS kicks in when the degree of overlapping is greater than one. The resulting degradation in performance is roughly the same, and is illustrated by the sharp increase in overhead for *LSS* osm-*LSS*, *TRAD* and osm-*TRAD*.

When the degree of overlap is greater than one, the transaction does not pay for allocation anymore, as the second transaction has already paid for it. This reduces the overhead for *TRAD* and osm-*TRAD*. As the degree of overlapping increases, the performance of LMs that keep track of locked objects (*TRAD*, osm-*TRAD*, *LSS*, osm-*LSS*) degrades significantly. In contrast, LMs that do not use lock book-keeping (i.e., all the variants of *NLSS*) scale better.

Again, memoization alone does not perform well because of the frequent invalidation of its cache due to redundant lock requests. This is particularly striking for T1, where 20% of lock requests result in a cache miss, and therefore, a call to the LM. Preceding the memoization cache probe with an OSM-test considerably improves the memoization cache efficiency, reducing cache misses to practically nothing. This makes osm-m-*NLSS* insensitive to the degree of overlapping because most lock acquisitions are serviced without calling the LM.

To sumarize, osm-m-*NLSS* outperforms all other LMs for the following reasons: (i) OSM tests filter out *all* of the redundant lock requests, which avoids unnecessary calls to the LM, and, most importantly, improves the efficiency of memoization caches; (ii) memoization reduces the number of calls to the LM for lock acquisition by 99.5%; and, (iii) lock state sharing without lock book-keeping makes lock release independent of the number of acquired locks. These properties reduces the locking overhead to under 70% for overlapping working sets.

9. Conclusions

This paper has studied the impact of several combinations of lock management method and lock barrier implementation on the processing overhead of automated locking in persistent programming languages using the PJama Virtual Machine as a case study. Our experiments indicates that the best performance are obtained with the NLSS variant of lock state sharing, a novel lock management method. NLSS dramatically reduces the memory consumption of locking by sharing lock data structures with identical values among locked objects, and avoiding the tracking of locked resource used in other lock management method to automatically release locks at transaction completion. The best performance of automated locking are obtained by combining NLSS with a two-stage lock barriers, which consists of an inconsistent lock ownership test to filter out all unnecessary and redundant lock requests, followed by, if the first stage fails, a memoization to avoid calling the lock manager. The resulting lock barrier cost between 8 (first stage only) to 14 (two stages) RISC instructions, and avoids calling the lock manager for 99% of the lock request issued by automated locking. This combination results in at least 30% performance improvement over other lock management methods, and makes automated locking scaling well with both the

size of the working set of a transaction and the degree of overlapping between transaction's working sets.

There is still room for improvement, as automated object locking imposes a considerable processing overhead: between 26% to 67% for an automated locking system based on the best variant of lock state sharing. This overhead is essentially due to the overwhelming number of unnecessary lock barriers. This clearly indicates the direction of future work toward fast and efficient program analysis methods to identify and eliminate unnecessary lock barriers during JIT compilation.

Trademarks Sun, Sun Microsystems, Sun Enterprise, Java, JDK and Solaris, are trademarks or registered trademarks of Sun Microsystems Inc. in the United States and other countries. SPARC and UltraSPARC are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries.

References

[1] Atkinson MP, Daynès L, Jordan MJ, Printezis T, Spence S. An orthogonally persistent Java^{TM}. *SIGMOD Record*, **25**(4), December 1996.

[2] Lindholm T, Yellin F. *The Java^{TM} Virtual Machine Specification, Second Edition*. The Java^{TM} Series. Addison Wesley, April 1999.

[3] Atkinson MP, Morrison R. Orthogonal persistent object systems. *VLDB Journal*, **4**(3), 1995.

[4] Daynès L, Atkinson MP , Valduriez P. Customizable Concurrency Control for Persistent Java. Chapter 7 in *Advanced Transaction Models and Architectures*, Jajodia S and Kerschberg L (ed), Data Management Systems. Kluwer Academic Publishers, Boston, 1997.

[5] Dieckmann S, Hölzle U. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. In Proc. of 13th European Conference on Object-Oriented Programming (ECOOP'99), Lisbon, June 1999, Springer Verlag.

[6] Carey M, DeWitt D, Naughton J. The OO7 benchmark. *ACM SIGMOD Int. Conf. on Management of Data*, Washington D.C, May 1993.

[7] Moss JEB. Working With Objects: To Swizzle or Not to Swizzle? *IEEE Transactions on Software Engineering*, **18**(8):657–673, August 1992.

[8] Haines N, Kindred D, Morrisset J, Nettles SM, Wing JM. Composing first-class transaction. *ACM Transactions on Programming Languages and Systems*, **16**(6):1719–1736, November 1994.

[9] Garthwaite A, Nettles S. Transactions for Java. *First International Workshop on Persistence and Java, Drymen, Scotland, September 1996.*

[10] Kim W, Ballou N, Chou H, Garza JF, Woelk D. Integrating an object-oriented programming system with a database system. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 142–152, September 1988.

[11] White SJ, DeWitt D. A performance study of alternative object faulting in pointer swizzling strategies. *18th Int. Conf. on Very Large Database*, 419–431, Vancouver, British Columbia, Canada, 1992.

[12] Carey M, DeWitt D, Franklin M, Hall N, McAuliffe M, Naughton J, Schuh D, Solomon M, Tan C, Tsatalos O, White S, and Zwilling M. Shoring Up Persistent Applications. In *ACM SIGMOD Int. Conf. on Management of Data*, 383–394, 1994.

[13] Versant. *Multi-thread Database Access*, May 1998. A Versant Application Note.

[14] White SJ, DeWitt D. QuickStore: a high performance mapped object store. *VLDB Journal*, **4**(4), 629–673, 1995.

[15] Biliris A and Panagos E. A High performance configurable storage manager. In *11th Int. Conf. on Data Engineering*, 35–43, Taipei, Taiwan, March 1995.

[16] Lehman T, Gottemukkala V. The design and performance evaluation of a lock manager for memory-resident database systems. In [27], 406–428.

[17] Gray J. Notes on data base operating systems. Volume 60 of *Lectures Notes in Computing Sciences*, 393–481. Springer-verlag, 1978.

[18] Gray J, Reuter A. *Transaction Processing : Concept and Techniques*. Morgan-Kaufman, 1993.

[19] Lomet D. Key range locking strategies for improved concurrency. *19th Int. Conf. on Very Large Database*, 655–664, Dublin,Ireland, 1993.

[20] Mohan C. Concurrency control and recovery method for B^+-tree indexes: ARIES/KVL and ARIES/IM. In [27], 248–306.

[21] Garcia-Molina H, Salem K. Main memory database systems: an overview. *IEEE Transactions on Knowledge and Data Engineering*, **4**(6):509–516, December 1992.

[22] Daynès L. Implementation of Automated Fine-Granularity Locking in a Persistent Programming Language. In *Software – Practice and Experience* **30**:1-37, 2000.

[23] Michie D. 'Memo' Functions and Machine Learning. *Nature*, (218):19–22, April 1968.

[24] Lewis B, Mathiske B, Gafter N. Architecture of the PEVM: A High-Performance Orthogonally Persistent Java^{TM} Virtual Machine. *9th International Workshop on Persistent Object Systems*, Lillehammer, Norway, September 2000.

[25] Daynès L, Gruber O, Valduriez P. Locking in OODBMS clients supporting nested transactions. *11th Int. Conf. on Data Engineering*, 316–323, Taipei, Taiwan, March 1995.

[26] The Standard Performance Evaluation Corporation (SPEC). *SPEC JVM98 Benchmarks* http://www.spec.org/osg/jvm98.

[27] *Concurrency Control Mechanism in Centralized Database Systems* Kumar V (ed), Prentice Hall, 1996.