Lightweight Flexible Isolation for Language-based Extensible Systems

Laurent Daynès, Grzegorz Czajkowski

Sun Microsystems Laboratories
901 San Antonio Road
Palo Alto, CA 94303,
USA
{Laurent.Daynes,Grzegorz.Czajkowski}@Sun.Com

Abstract

Safe programming languages encourage the development of dynamically extensible systems, such as extensible Web servers and mobile agent platforms. Although protection is of utmost importance in these settings, current solutions do not adequately address fault containment. This paper advocates an approach to protection where transactions act as protection domains. This enables direct sharing of objects while protecting against unauthorized accesses and failures of authorized components. The main questions about this approach are what transaction models translate best into protection mechanisms suited for extensible language-based systems and what is the impact of transaction-based protection on performance. A programmable isolation engine has been integrated with the runtime of a safe programming language in order to allow quick experimentation with a variety of isolation models and to answer both questions. This paper reports on the techniques for flexible fine-grained locking and undo devised to meet the functional and performance requirements of transaction-based protection. Performance analysis of a prototype implementation shows that (i) sophisticated concurrency controls do not translate into higher overheads, and (ii) the ability to memoize locking operations is crucial to performance.

**Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002**

1. Introduction

Dynamically extensible systems, such as Web browsers, Web servers, mobile agent platforms, and application servers, are characterized by their ability to dynamically download programs that might interact with locally installed trusted components and with one another. The Java™ programming language [GJS+00] has been the main actor in the development of such systems due to a combination of several features. The language is network-centric and includes programmable built-in security mechanisms that programmers can use to control the scope of interactions with other programs. The runtime environment of the language consists of a virtual machine executed by a single operating system (OS) process; the virtual machine in turn executes applications and their dynamic extensions. Co-locating programs within the same address space benefits scalability and performance. Protection between executing programs is enforced via software mechanisms that leverage language safety to provide a flexible and efficient alternative to hardware-based protection mechanisms [BSP+95].

The original approach to protection [Gong99], based on class loaders, separate name spaces, and security managers, provides only a partial isolation of components from one another and serves well only within the limit of a set of rules that co-located programs must conform to. This is insufficient to allow arbitrary extensions. Increasing the level of component isolation in the Java programming language and other similar settings has been a focus of various projects [HCC+98,BR00]; the more promising ones achieved isolation via re-architecting the virtual machine to provide safe and scalable multitasking [BHL00,CD01]. These designs address many problems present in currently deployed extensible middleware but are usually strongly influenced by pragmatic concerns mixed with a fair dose of conservatism inherent in large software projects. Typically, the notion of protection

domains is introduced; references to objects cannot leave a protection domain; instead, object copies or revocable remote references are passed across domains.

Such an approach to protection makes programming of extensible systems where objects may need to be simultaneously accessible to multiple programs cumbersome. Programming becomes particularly difficult when objects shared across protection domains are mutable, as copies need to be continuously transmitted between the interacting parties, and consistency issues may need to be dealt with. Finally, copying increases the latency of inter-component communication and may degrade performance noticeably.

Alternatives to this model promote safe direct object sharing and circumvent problems due to object aliasing by decoupling naming from method invocation, and by introducing access control on the latter [BR00, RSC92]. Thus, object references can freely cross protection domains and be stored into arbitrary locations while method invocations remain subject to access control.

All these approaches neglect a crucial aspect of isolation: fault containment. Prohibiting, or restricting, object sharing via the means of the type system, capability mechanisms, or access control, does not prevent fault propagation from one component to another. Data of a component may have been left partially modified by another trusted component that has failed to complete all its interactions. The infected component can in turn propagate the failure further. The only remedy to this is to kill all infected components, but that assumes that the original failure can be detected, and the infected components repaired. In absence of such capabilities, the only resort is to restart the entire system.

We advocate borrowing the notion of transactions from database systems and adapting it to serve as protection domains in order to address both access control and fault containment. The rationale is the observation that in many advanced transaction models pioneered for non-traditional database applications transactions can be organized so that access to a set of objects can be completely prohibited to some transactions, while other transactions are allowed to compete for them. This suggests that the machinery underlying concurrency control can also be exploited to realize access control at no additional cost.

In this approach, transactions act as protection domains: every object is *owned* by only one transaction, which is responsible for authorizing access to the object, and every program executes as a transaction. As in other safe direct object sharing approaches, naming is decoupled from method invocation. The difference is that invocations are subject to concurrency control, which simultaneously enforces isolation and access control. A violation of either leads to a conflict. Access violation conflicts force the offender to undo its actions.

Transaction-based protection also deals with safe termination, which is otherwise difficult to achieve [RCW01], and with safe concurrency (the ability to preserve a consistent state when executing arbitrary code against shared data.) The combination of safe termination, access control, fault containment, and safe concurrency makes transaction-based protection a compelling foundation for extensible language-based systems.

Exactly what transaction models are well suited for transaction-based protection, and how well such an approach compares with others on performance grounds are the two research issues on our agenda. In order to formulate answers to these questions, we have adopted a design that enables quick prototyping of various isolation models. This paper reports on our efforts to build a flexible transactional isolation engine with performance compatible with the requirements of an extensible language-based system, and demonstrates this goal is feasible based on a detailed analysis of the performance obtained with various isolation models.

A number of characteristics set our work apart from other attempts to integrate transactional features with programming languages, such as efforts related to persistent programming languages (PPLs) [LCJ+87, HBM93] and object-oriented database management systems (OODBMS). The architecture often adopted in these settings is distributed: either client/server or peer-to-peer and each application executes in separate OS process. A substantial part of the design is driven by the requirement to efficiently ensure durable changes to persistent data and to reduce network latencies. In contrast, transactional control in our system is geared toward safe sharing of volatile data that are directly accessible by applications, which all run in the same address space and share the run-time system. Many of the techniques described in this work are applicable to PPLs and to database systems. In particular, the following contributions are relevant to the database community:

- a space-efficient, high-performance, fine-granularity flexible lock management technique suitable for main-memory resident systems.
- an analysis of the space and processing overheads incurred by the concurrency control of a variety advanced transaction models.
- evidence that a more sophisticated concurrency control does not translate into higher overheads.

The paper is organized as follows. Section 2 contains an overview of our experimental architecture for transaction-based protection. Section 3 describes the main features of the flexible isolation engine and examples of its usage. Section 4 covers the implementation techniques used to efficiently support flexible locking, and Section 5 analyses their performance. Section 6 contrasts the contributions of this paper with related work.

2. System Overview

Our prototype platform for experimenting with transaction-based protection provides an alternative

platform for the Java programming language with automatically enforced flexible isolation mechanisms. Isolation is added without changing the language definition. This allows the platform to execute existing programs without any modifications to their source or compiled forms, regardless of the chosen isolation model.

The platform is realized by augmenting an implementation of the Java virtual machine (JVM™) with a programmable isolation engine. The engine supplies flexible locking and undo mechanisms. It provides two APIs: the internal API, used mainly by the interpreter and the dynamic compiler to automate the interaction with the isolation engine, and the external API, exposed to (expert) programmers as a package written in the Java programming language. The external API enables relatively simple programming of new behaviors of the isolation engine. It also includes an isolation manager that mediates all requests to the isolation engine. The isolation manager itself is a special transaction that lives as long as the virtual machine. This design leverages the isolation mechanisms to protect the isolation manager's data structures from misbehaving programs.

Transactions are instances of a class that encapsulates entry points to the isolation manager. Entry points are similar to the trap mechanism of an OS: they switch to the context of the isolation manager transaction prior to executing the corresponding "kernel" service, and exit back to the original transaction context.

In addition to entry points to begin, commit, and abort a transaction, two other entry points are defined. The first one initiates a thread to execute a program in the context of the transaction. The second entry point sends an event to a programmer-defined extension that customizes the behavior of the isolation engine. This is used to extend the default transaction interface (e.g., when augmenting the transaction interface with split and join methods).

From the point of view of application programmers, executing an arbitrary program under a particular isolation behavior is fairly simple: a transaction object is instantiated, and its methods are invoked to successively begin the transaction, launch the execution of a program in the transaction, and terminate the transaction. Any programs normally executable by the JVM can be executed by a transaction, including multi-threaded ones.

The programming interface to the isolation engine is similar to the design presented in a previous work on flexible transaction management for PPLs [DAV97] and will not be further described here.

3. Flexible Isolation Engine

Frameworks for specifying transaction models such as ACTA [Chry90] have helped to identify new primitives general enough to express the concurrency control of numerous advanced transaction models. Our flexible isolation engine builds on these results. It is not intended to be general; rather, it aims at a family of transaction models that we believe address well isolation and safe sharing in language-based extensible systems. It is based on the observation that locking protocols differ in: (i) the number of lock ownerships required by each transactional entity, (ii) the conflict detection mechanism, (iii) the conflict resolution mechanism, and (iv) how transaction models use the notion of delegation [Chry90].

3.1 Locking contexts

The number of distinct lock ownerships required by each transactional entity to realize concurrency control differs between transaction models. A transaction needs only one type of lock ownership in the classic ACID transaction model, and two in the nested transaction model (one for held locks, and one for retained locks [HR93]). The concurrency control of the Apotram transaction model [Anfi97] can be expressed using a single type of lock ownership per transaction, and three other types of lock ownership per sub-databases. Generally, a single lock ownership is used to acquire the locks a transaction needs to perform its operations. Additional lock ownerships are used as rings of protection that prevent a specific set of transaction from doing certain operations. How many lock ownerships are used depends on the complexity of the rules that determine what transaction should be denied access. Based on these observations, the isolation engine separates lock ownership from the transactional entity and lets programmers specify how they are associated.

The flexible isolation engine represents lock ownership with a locking context. Locking contexts are either active or passive. Locking contexts can transfer locks to other locking context (see delegation of locks below) or release them. Locks can only be acquired on demand with an active locking contexts: transactions assign them to the threads running on their behalf; the runtime automatically acquire locks using the locking context of the current thread.

The behavior of a locking context with respect to conflict detection, resolution, and notification, as well as transferring of lock ownership can be programmed via a simple interface. The salient features of this interface and their effect on the locking logic are described below.

3.2 Ignore-conflict relationships

Conceptually, the state of a lock consists of one set of lock owners per lock mode. *Owners(l,m)* denotes the set of locking contexts that own lock *l* in mode *m*. A function *Compatible(m_1,m_2)* determines whether lock mode m_1 is compatible with lock mode m_2. Compatibility of lock modes is defined by the commutability of the corresponding operations. Incompatibility of modes is considered a conflict. Conflicts are denoted m_1/m_2, where m_1 is the requested mode, and m_2 a mode incompatible with m_1. When using read (*r*) and write (*w*) lock modes only, lock ownership is expressed with a pair *<Owners(l,r), Owners(l,w)>* of owner sets. Three types of

conflict can occur: read/write (*r/w*), write/read (*w/r*), and write/write (*w/w*).

The conflict detection mechanism of a lock manager can be generalized by specifying ignore-conflict relationships between locking contexts. An ignore-conflict relationship alters the default evaluation of conflicts by selectively ignoring incompatible owners of requested locks. For instance, a locking context C_1 may specify a relationship with a locking context C_2, such that all conflicts (*r/w, w/r, w/w*) with C_2 are ignored when deciding whether C_1 can be granted a lock.

An ignore-conflict relationship involves two locking contexts, one of which must be an active locking context; passive locking contexts cannot generate conflicts since they cannot request a lock. An active locking context can choose to ignore conflicts with passive and active locking contexts. We use $\lambda(C_i \rightarrow C_j, m_1/m_2)$ to denote an ignore-conflict relationship that allows an active locking context C_i to ignore a conflict of type m_1/m_2 with a locking context C_j, where m_1/m_2 can be any of *r/w, w/r, w/w*. The isolation engine allows only symmetric ignore-conflict relationships between two active locking contexts. A symmetric ignore-conflict relationship for a conflict m_1/m_2 is equivalent to the two asymmetric ignore-conflict relationships $\lambda(C_i \rightarrow C_j, m_1/m_2)$ and $\lambda(C_j \rightarrow C_i, m_2/m_1)$.

The ignore-conflict relationships of an active locking context *C* are used to derive ignore-conflict-with (*ICW*) sets of locking contexts with which conflicts can be ignored. For a given conflict type m_1/m_2, the set is defined as $ICW(C, m_1/m_2) = \{C_k: \lambda(C \rightarrow C_k, m_1/m_2)\}$.

Given these definitions, a request for a lock *l* in mode *m* by a locking context *C* creates a conflict if and only if the following condition is false:

$\forall\ m_i,\ Compatible(m, m_i) \vee (\neg Compatible(m, m_i) \wedge (Owners(l, m_i) \subseteq ICW(C, m/m_i) \cup \{C\}))$

Conflicts are resolved either by blocking the requester or by notifying a third party so that a custom action can be triggered (e.g., abort, negotiation of additional ignore-conflict relationships, etc.). In particular, conflict notifications are useful to achieve the "rollback on access violation" policy of transaction-based protection.

3.3 Delegation of locks

Advanced transaction models often employ some form of lock ownership transfers, also referred to as ***lock delegation***. Delegation operations transfer the ownership of a lock from one set of locking contexts, referred as the *delegators*, to another set of locking contexts, referred as the *delegatees*. The sets of delegators and delegatees involved in a delegation operation are often singletons, and the most common situation is to delegate at once all the locks of the delegator(s). Other flavors include delegating one or more specific locks. Delegations to multiple locking contexts are allowed only if the multiple delegatees do not conflict with one another; otherwise, a deadlock situation is created.

3.4 Examples

The mechanisms just described were used to realize the concurrency control of several transaction models with features that are appealing to transaction-based protection.

The strictest form of isolation is guaranteed by conflict serializability (CSR). Realizing CSR is straightforward: each transaction *T* is associated with a single active locking context *C*, set up so that no conflicts can be ignored (i.e., $\forall\ m_i/m_j,\ ICW(C,m_i/m_j)=\emptyset$). Upon transaction abort or commit, all of *C*'s locks are released.

The nested transactions model [HR93] offers a more general execution model that may be better suited for a language-based extensible system. The model resembles the standard OS process model and provides a similar tree-structured control flow, but with stronger guarantees with respect to failures and concurrent access.

Lock-based implementations of the concurrency control of nested transactions employ, in the general case where both parent-child and sibling parallelism is supported, two types of lock ownership per transaction: held and retained locks. Held locks are locks acquired by a transaction to perform its own operations. Retained locks are locks that were delegated to a transaction by its committed sub-transactions. Retained locks are inherited by their retainer and by its inferiors [HR93] in the transaction hierarchy. When a transaction commits, it delegates its held and retained locks to its parent. When it aborts, it releases its held and retained locks.

These locking rules translate into ignore-conflict relationships and delegation operations over locking contexts as follows. Each transaction *T* is associated with two locking contexts: an active locking context $C_h(T)$ and a passive one $C_r(T)$, which correspond, respectively, to the held and retained types of lock ownership. Ignore-conflict relationships for a top-level transaction T_t and a sub-transaction T_s are set up as follows:

$\forall\ m_i/m_j,\ ICW(C_h(T_t), m_i/m_j) = \{C_r(T_t)\}$

$\forall\ m_i/m_j,\ ICW(C_h(T_s), m_i/m_j) =$

$\{C_r(T_s)\} \cup ICW(C_h(Parent(T_s)), m_i/m_j)$

where *Parent(T)* denotes *T*'s parent transaction. Upon an abort of a transaction *T*, $C_r(T)$ and $C_h(T)$ release their locks. Upon commit of a transaction *T*, $C_r(T)$ and $C_h(T)$ delegate their locks to $C_r(Parent(T))$ if *T* is a sub-transaction; otherwise, they both release their locks.

Intuitively, $C_r(T)$ disallows any transaction outside of the hierarchy rooted by *T* from doing operations that are incompatible with those already carried out by committed sub-transactions of *T*, while allowing those inside the hierarchy, including *T* itself, to do so, provided they held the appropriate locks. Note also that ignore-conflict relationships are set only between active locking contexts (e.g., $C_h(T)$), which are used to acquire the locks of their respective transactions, and passive locking contexts (e.g., $C_r(T)$), which are used to retain locks of committed sub-transactions Thus, no transaction can ignore-conflict with a lock acquired (as opposed to just retained) by another.

The Apotram transaction model [Anfi97] also enables behavior that is of interest to transaction-based protection. It addresses the needs of collaborative applications with two new correctness criteria: *nested conflict serializability* and *conditional conflict serializability* (CCSR), which translate in practice into two mechanisms for concurrency control: nested databases and parameterized lock modes. They can be combined or used independently. Because nested databases are reminiscent of nested transactions, we only discuss CCSR and parameterized lock modes.

CCSR weakens CSR in an application-controlled manner by allowing read-write and write-read pairs of operations to conflict conditionally, while write-write operations always conflict. CCSR can be achieved by determining the compatibility of lock modes conditionally: the condition is specified as a predicate over parameters to lock modes. Let *r(A)* and *w(B)* denote a parameterized read and write mode, respectively, where *A* and *B* denote subsets of some parameter domain *D*; *r(A)* and *w(B)* conflict unless $B \subseteq A$. Non-parameterized modes *r* and *w* correspond to the parameterized modes *r(∅)* and *w(*)*, such that $\forall D, * \supseteq D$. It follows that *r(∅)* is incompatible with all write modes and *w(*)* is incompatible with all read modes.

Applications use parameterized lock modes by defining domains of parameter values and assigning sets of such values to transactions. The following example illustrates a typical use of this mechanism. Two authors Alice and Bob want to collaborate on a document such that each of them can read but not overwrite what the other is modifying. They also want to prevent anyone else from seeing their changes or from changing anything they have read. A single parameter value suffices to cover these needs. The authors must define a single-valued domain they would keep confidential. Let us call this domain $D = \{\alpha\}$. To achieve the behavior they want, the authors run their transactions using the parameterized lock modes $r(\{\alpha\})$ and $w(\{\alpha\})$. The following history of operations is then allowed:

$H = <r(T_B, O_1), r(T_B, O_2), w(T_B, O_1), r(T_A, O_1), r(T_A, O_2), w(T_A, O_2)>$

where *x(T, O)* denotes an operation *x* performed by *T* on an object *O*, and T_B (respectively, T_A) denotes Bob's (Alice's) transaction. After T_B locked O_1 in read mode parameterized with $\{\alpha\}$, no other transactions can write O_1 except T_A, but any transaction can read O_1. After T_B locked O_1 in write mode, no other transactions can read O_1 except T_A.

Now, let us assume that the manager of Bob and Alice wants to observe their progress as well as that of another team. Both teams require isolation from each other, and the manager requires strict isolation on his own work. In order to achieve this behavior, the domain *D* is extended to include another parameter value, β so that $D = \{\alpha, \beta\}$. The following specification leads to the desired behavior: transactions issued by members of the first (second) team use the parameterized lock modes $r(\{\alpha\})$ and $w(\{\alpha\})$ ($r(\{\beta\})$ and $w(\{\beta\})$); the manager's transactions use $r(\{\alpha,\beta\})$ and *w(*)*.

The effect of parameterized lock modes is obtained by associating each transaction with a single locking context and setting symmetric *r/w* and *w/r* ignore-conflict relationships as described below. Using *r(A)* to acquire read locks is equivalent to ignoring *r/w* conflicts with the locking contexts of all transactions that use a subset of *A* to acquire their write locks. Conversely, using *w(B)* to acquire write locks is equivalent to ignoring *w/r* conflicts with the locking context of all transactions that use a superset of *B* to acquire their read locks. More formally, let *ρ(C,r)* and *ρ(C,w)* each denote the set of parameter values used by a transaction associated with a locking context *C* to parameterize the read and write lock mode. The following defines a conversion of a specification of parameterized lock mode usage into an equivalent set of ignore-conflict relationships:

$\forall C, ICW(C, r/w) = \{C_k: \rho(C_k,w) \subseteq \rho(C,r)\}$

$\forall C, ICW(C, w/r) = \{C_k: \rho(C_k,r) \supseteq \rho(C,w)\}$

An implementation of parameterized lock modes based on ignore-conflict relationships consists essentially of maintaining a data structure that maps sets of parameters to the transactions that use these sets. The map helps to determine ignore-conflict relationships that must be set when a transaction is assigned a set of parameters. When a transaction completes (either commits or aborts), it is removed from the list of transactions of the parameter sets used by that transaction.

3.5 Technical Challenges

In order to enforce isolation, every program must behave as a well-formed transaction regardless of what isolation model is used. Well-formedness means that transactions execute an operation on an object only when they own the lock of that object in the mode corresponding to the operation. Further, if the operation updates, proper undo information must have been recorded before applying the updates. Automation of these tasks avoids depending on the programmer to always formulate well-formed transactions and simplifies the programmer's work.

Automated object locking is accomplished by transparently planting in programs small sequences of instructions, called *lock barriers*. They issue lock requests to the lock manager of the isolation engine, and may trigger logging activity. Augmenting a high-performance implementation of the JVM with these mechanisms while incurring minimal performance impact is challenging.

Accessing an object consists of a main-memory access as most implementations of the JVM avoid explicit null pointer checks and instead rely on segmentation fault signal handling. Array bound checks, required by the language specification, are harder to avoid and typically add three instructions. In both cases each instruction added by a lock barrier to an access path has a noticeable

performance consequence. The code for lock barriers should therefore be extremely lean.

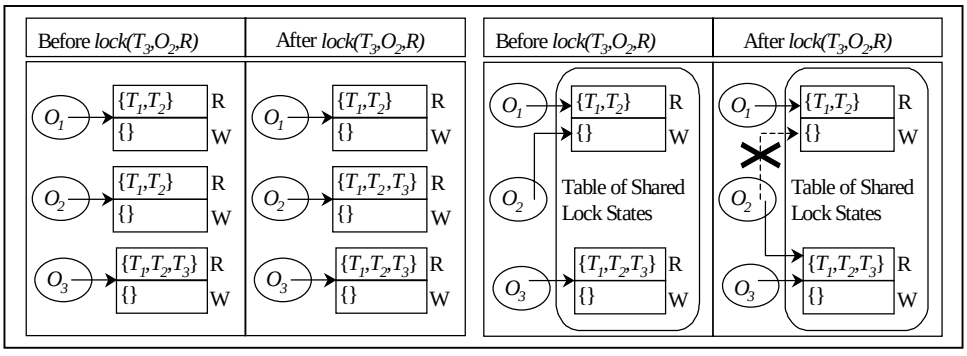
Ideally, a transaction needs to request the lock of an object it accesses only once, before its first access to that object. Identifying ahead of time the first access to an object by an arbitrary program is impossible in general. A pragmatic solution is to precede every object access with a lock barrier, and to rely on compiler analysis to identify and remove as many redundant lock barriers as possible. Because compilations take place at runtime, the amount of analysis that can be done must be limited in order to reduce the impact on performance. Dynamic class loading complicates analysis further as it can invalidate past decisions taken by the optimizer. Both issues must be considered when eliminating redundant lock barriers.

Another challenge is posed by the small size of objects, typically between 16 to 42 bytes and by a large number of individual objects accessed by programs [DH99], when compared to a typical database transaction. This requires space-efficient lock management that scales well with the number of locks.

Locking is traditionally implemented by associating a resource with a data structure that represents the lock protecting that resource. The data structure, which we refer to as ***lock state*** throughout this paper, typically specifies what transactions own the lock and in what mode, along with pending requests for the lock. Locking operations first locate the lock state that represents the lock of a resource, and then update it as needed to reflect the effects of the operation.

This traditional approach does not scale in terms of memory consumption because it requires as many lock states as locks and also because of the extra data structures required to keep track of the locks of each transaction in order to automatically locate and release them all upon transaction termination. Database systems have over several decades engineered a well-tuned battery of locking protocols that help in circumventing the poor scalability inherent to this approach. These techniques heavily depend on a well-identified hierarchy of data containers and on the associative nature of database programming languages. This makes them a poor match for systems that tightly couple a transaction-processing engine with a general-purpose programming language.

Figure 1: Lock State Sharing.



4. Lock State Sharing

Lock state sharing is a novel lock management method that makes the space overhead of locking small and independent of the number of locked resources [DC01], and does so without depending on any particular locking protocol. However, the method reported in [DC01] only supports strict isolation. This section presents extensions to support delegation and programmable conflict detection as described in Section 3. Background material on lock state sharing is covered first.

4.1 Principles

Lock state sharing relies on the observation that, at any time, the total number of distinct values of locks in a system is very small compared to the total number of locked resources. In other words, one can expect many locks to have the same value. A lock manager implementation can take advantage of this situation by representing locks of equal value with the same lock state, instead of with a different lock state each. Then, instead of updating lock states as traditional lock management methods do, locking operations change the association between a resource and its lock state.

Shared lock states representing locks of currently locked resources are recorded in an associative table of shared lock states (TLS) keyed on lock values. The TLS does not hold all possible lock values: it is initially empty, and shared lock states are added to it as needed, i.e., when no already existing shared lock state holds a lock value needed to represent the lock of a resource. Garbage collection techniques determine unused shared lock states and remove them from the TLS.

Figure 1 illustrates how lock state sharing works by comparison with a traditional approach to locking. 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 locks is shown before and after the acquisition of O_2's read lock by T_3. In the traditional approach, each object is associated with a private lock state and T_3's request for O_2's lock would be processed by updating O_2's private lock state to reflect the lock's new value (i.e., lock owned in read mode by T_1, T_2 and T_3). Using lock state sharing, O_1 and O_2 share the same lock state, which holds the value that the two private lock state representing their respective locks would have. Instead of updating the lock state associated with O_2 to process T_3's request, the lock manager scans the TLS for a shared lock state

holding the new value of O_2's lock, and atomically changes the shared lock state associated with O_2 with the one returned by the TLS.

It is crucial to understand that in both approaches, every object is an independent unit of locking: the sharing of lock states of equal value must not be mistaken with protecting several objects using the same lock.

Lock state sharing enables three other important optimizations [DC01]: (i) it makes the tracking of locked objects (i.e., of acquired locks) unnecessary, which results in dramatic space savings and, together with the sharing of lock state, contributes to making the space overhead of locking independent of the number of locked objects; (ii) it makes the implementation of bulk locking operations independent of the number of locked objects involved; and (iii), it allows the use of memoization [Mich68] for all individual locking operations. As will be shown later, adding delegation and programmable conflict detection do not impact these optimizations.

4.2 Bulk locking operations

Bulk locking operations apply to ***all*** the locks of a given owner. The release of all the locks of a transaction upon its termination is an example of a bulk locking operation. Delegation of all the locks of a sub-transaction to its parent in a nested transactions model is another example. Traditional lock management methods explicitly keep track of all locks acquired by each lock owner in order to locate automatically all its locks.

Since the number of shared lock states is typically very small, the lock manager can locate all the shared lock states that encode the value of locks owned by a given locking context just by scanning the TLS. The effect of a bulk locking operation is obtained by updating the value of all the shared lock states found. When bulk-releasing the locks of a locking context C, the update consists of removing C from all the owner sets of these shared lock states. This may turn some shared lock states into duplicates of others already recorded in the TLS. In this case the modified lock state is moved to a list of duplicates. Duplicates that pre-existed the bulk locking operations are scanned for lock states encoding locks of the requester in order to update them too. Pointers to duplicates stored in object headers are opportunistically replaced with their original during garbage collection in order to accelerate their reclamation.

Because a shared lock state can potentially represent the value of many locks, changing its value changes the values of all such locks. Updating a shared lock state is only used to implement the effect of a lock operation that should affect all the locks represented by that lock state, which is the case for bulk locking operations.

4.3 Memoized lock operations

Although lock state sharing adequately addresses space overhead, the cost of lock acquisitions remains prohibitive for the runtime of a programming language. Acquiring a lock takes four steps:

- determine the absence of conflicts, possibly using the ignore-conflict relationships of the requester,
- compute the new value the locking operation produces from the value of the lock state currently associated with the object protected by the lock,
- retrieve from the TLS the shared lock state that encodes the new value (if none is found, the TLS automatically creates one),
- atomically change the shared lock state currently associated with the object to the retrieved one.

The first three steps are performed without synchronization with concurrent threads. Instead, the last step verifies that the value of the object's lock has not changed before setting its new value. Each object includes in its header a pointer to the shared lock state that encodes the value of its lock. In this case, the last step consists of a single atomic compare-and-exchange of pointers to shared lock states. If this exchange fails, the four steps are repeated again.

A simple form of memoization [Mich68] can be used to substantially speed up lock acquisition by eliminating the first three steps of lock acquisition for most requests. The idea is to maintain, per thread, a small cache that remembers what pointers to shared lock states were exchanged by past granted lock requests. Such caches remember four-tuples $<C, l_i, l_f, m>$ such that $l_f = f(C, l_i, m)$, where C is a locking context, m is a lock mode, l_i is a pointer to a shared lock state, and f returns the pointer to a shared lock state that represents the value that a lock of initial value l_i must have after C's request is granted.

4.4 Flexible locking primitives

In order to support the features of the flexible isolation engine described in Section 3, lock management based on lock state sharing must be extended (i) to enable programming of the conflict detection logic of a locking context based on ignore-conflict relationships, and (ii) to allow delegation of one, several, or all the locks of one or more locking contexts to one or more others.

Extending the conflict detection logic with ignore-conflict relationships augments each locking context with two sets of locking contexts per type of conflict. For a locking context C and a conflict of type m_1/m_2, these sets are denoted, respectively, $ICW(C, m_1/m_2)$ and $UICW(C, m_1/m_2)$. $ICW(C, m_1/m_2)$ records the set of locking contexts with which C ***can*** ignore m_1/m_2 conflicts, whereas $UICW(C, m_1/m_2)$ records the set of locking contexts with which C ***has*** ignored one or more m_1/m_2 conflicts to obtain a lock (hence, $UICW(C, m_1/m_2) \subseteq ICW(C,m_1/m_2)$). This distinction helps to avoid expensive synchronization between conflict detection and updates to ignore-conflict relationships. For a given lock request, conflicts are initially evaluated using the *UICW* sets of the requester, without synchronization. If conflicts remain, the monitor

protecting the *ICW* sets is obtained, and conflicts are re-evaluated using the *ICW* sets. If no conflicts are found, the members of *ICW* not in *UICW* that permitted conflicts to be ignored are added to *UICW*.

The use of ignore-conflict relationships has little impact on how locking operations are performed with lock state sharing. In particular, it does not disallow the use of memoization. The caches used to memoize lock acquisitions record the input and output values of lock requests that do not cause conflicts, including those requests that have ignored conflicts because of existing ignore-conflict relationships. Changing the ignore-conflict relationships of a locking context does not invalidate the cached results of memoized operations since conflicts that have already been ignored before the changes will still be ignored after: adding ignore-conflict relationships only increases the number of conflicts that can be ignored, and removal of ignore-conflict relationships is only allowed for relationships that have not been used or are not used anymore (i.e., all the locks acquired using them have been either released or delegated).

Delegation of the lock of a single resource is realized like all other locking operations. The value the lock should have as a result of delegation is first computed by applying the effect of the delegation on a copy of the shared lock state currently representing the resource's lock. The new lock value is computed by removing the delegator(s) and adding the delegatee(s) to each of the owner sets including at least one of the delegators. The value obtained is then used to search the TLS for a shared lock state encoding it, which is then set using an atomic compare-and-exchange instruction. Memoization can also be applied to delegation. Similarly to lock release, lock delegation may take a bulk form, that is, all the locks owned by a locking context may be delegated at once. Bulk lock delegations differ from bulk lock release only by the transformation applied to shared lock states.

4.5 Implementation details

Sets of locking contexts are central to many aspects of the extension to lock state sharing described above. The representation of shared lock states consists mainly of owner (i.e., locking context) sets. Conflict detection consists of one or more set inclusion tests between owner sets and set of locking contexts representing ignore-conflict relationships.

Using fixed-size bitmaps to represent sets of locking contexts has two advantages. First, performance-critical set operations translate into efficient bitmap manipulations. This leads to a fast implementation of lock ownership test, which is the most critical operation for the overhead of automated locking. Second, it vastly reduces the complexity of the lock manager, in particular with respect to synchronization and memory management. Because the total number of shared lock states maintained in a system is small, the overcapacity in space resulting from using fixed-size bitmaps is not a concern, and sizes of up to several hundred bits can be easily afforded.

Class instances and arrays have been modified to include a pointer to a shared lock state in their header. Upon object allocation, this pointer is set to the address of the shared lock state encoding the value of write locks owned by the creator of the object only.

Both the interpreter and the dynamic compiler of the JVM have been modified to support automated locking. The bulk of the effort went into changing the dynamic compiler, since interpreted code has relatively little impact on performance. The interpretation of bytecodes that read or write the mutable part of an object, or invoke one of its methods, has been modified to execute a lock barrier beforehand. The dynamic compiler has been made aware of lock barriers, and simple optimizations to avoid generating unnecessary lock barriers have been incorporated. These optimizations are only local to basic blocks of the control flow graph. More sophisticated optimizations (e.g., hoisting lock barriers out of loop, simple escape analysis) are still under development.

A lock barrier, both interpreted and generated by the dynamic compiler, consists of three successive stages. The first stage filters redundant lock requests by testing if the requester already owns the lock in the mode requested. Most lock requests exit the barrier at this stage. The second stage performs an inline memoized lock operation to acquire the lock. If memoization fails, control is transferred to the runtime to call the lock manager to process the lock request and to update the memoization cache accordingly. On an UltraSPARC™ processor, the first stage consists of 7 instructions; the second stage adds between 6 to 16 instructions depending on how many lines of the memoization cache are used before memoization succeeds (up to 3).

Undoing of updates is based on physical logging, i.e., log records hold the value of data items before their modification. Programs written in the Java programming language tend to generate a population of very small objects [DH99]. Their mutable part is often even smaller. This makes the recording of the whole mutable part of an object upon the very first update to it an interesting strategy: it avoids the use of an additional write barrier in the access path to objects. Instead, the existing lock barrier triggers logging upon acquisition of a write lock. Each locking context is associated with an undo log made of one segment per thread executing with that locking context. Records generated by one thread are written to its dedicated segment of the log.

The above strategy would be too expensive for large arrays. Logging must be done sparsely in this case, which requires an additional write barrier to detect whether the portion of the array about to be updated has been logged already. To this end, large arrays are allocated in a special section of the heap subject to a variant of card marking [HBM93]. Cards are systematically tested. Unmarked cards are recorded to the log, then marked.

5. Experiments

Good performance is a necessary condition for the acceptance of transaction-based protection in language-based extensible systems. Overheads should be tolerable when the ability to share objects directly is taken advantage of; when no sharing takes place, the overheads should be as low as possible.

In order to explore these issues we have extended the Java HotSpot™ virtual machine [Sun00] version 1.3.1, JDK version 1.3.1, with the flexible isolation engine, described in Sections 3 and 4. From now on, we will refer to our prototype as TPVM, and to the original Java HotSpot virtual machine as HSVM.

Although we are still debating which transaction model is best suited for transaction-based protection, the candidate will likely incorporate features inspired by the nested transactions and Apotram models. To understand their impact on performance, we have implemented extensions of the flexible isolation engine for the nested transactions model (NT) and for a variation of the Apotram transaction model that realizes parameterized lock modes for flat transactions (APT).

program	overhead
jack	18%
javac	20%
db	16%
mtrt	28%
compress	59%
mpeg	39%

Table 1: overheads for SPECjvm98.

TPVM supports three types of transactions by default: the isolation manager's ever-lasting transaction; single-level ACID transactions; and a special type of ACID transaction, called *main* transactions, which are capable of initiating transactions of arbitrary type as independent transactions. When TPVM starts, it first initializes system classes in the context of the isolation manager's transaction, which then launches a main transaction to execute the program specified in the command line. The program typically waits for incoming request to execute other specified programs with a particular type of transaction (e.g., ACID, NT or APT).

We ran a series of experiments to compare TPVM against HSVM. The SPECjvm98 suite of benchmarks [Spec98] was used to evaluate the overhead of transaction-based protection in situations where data are not shared. All other experiments use a main-memory implementation of the OO7 benchmark [CDN93] as an example of shared data. Same code is used for all experiments, regardless of whether HSVM or TPVM is running it, and regardless of transaction model used. All measurements were carried out on a Sun Enterprise™ 420R server with 4 UltraSPARC™ IIi processors clocked at 450Mhz, with 4GB of main-memory, running the Solaris™ Operating Environment version 2.8.

Table 1 shows the overhead that TPVM adds when programs of the SPECjvm98 suite are executed as ACID transactions. Overheads range from 16% (db) to 59% (compress), and are caused by redundant lock requests only. Programs with overheads above 20% are array and loop-intensive. This reflects the current lack of compile-time analysis in TPVM to detect and eliminate redundant lock barriers against arrays or within loops.

5.1 Cost of flexible isolation

Figure 2 and 3 reports the overhead of executing the traversal operations t1, t2a, and t2b defined in the OO7 benchmark against a small database configuration with different isolation models, namely ACID, NT and APT. Each of these models exercises a different set of features of the flexible isolation engine. ACID does not use any of the programmable features. APT only uses ignore-conflict relationships, while NT uses both ignore-conflict relationships and delegation. The operations traverse exactly the same number of objects and acquire the same number of read locks, but update different number of objects (none for t1, 493 for t2a and 9860 for t2b), resulting in different number of write locks and amount of logging. Regardless of the transaction model used, and given a traversal operation, every transaction issues the same number of lock requests, acquires the same number of locks, and logs the same amount of data. Typically, 3×10^7 lock requests are issued for a single traversal operation, 98.5% of which are redundant.

In both figures, the isolation overhead is broken down into overhead due to redundant lock requests (i.e., requests for a lock that the requester already owns) and the overhead imposed by the mechanisms that actually enforce isolation (namely lock acquisition, bulk release or delegation at transaction termination, undo logging, and transaction management operations, which may include calls to the flexible isolation engine to program its behavior, such as, setting ignore-conflict relationships). The overhead eliminated by memoization alone is also reported (topmost component of every bar).

Figure 2 reports the overhead of isolation when shared data are not access concurrently (the measured transaction operates alone on the shared data). In this case, the overhead of locking is minimal since no conflict detection is performed, and the programmed part of conflict detection isn't exercised. Furthermore, lock requests to

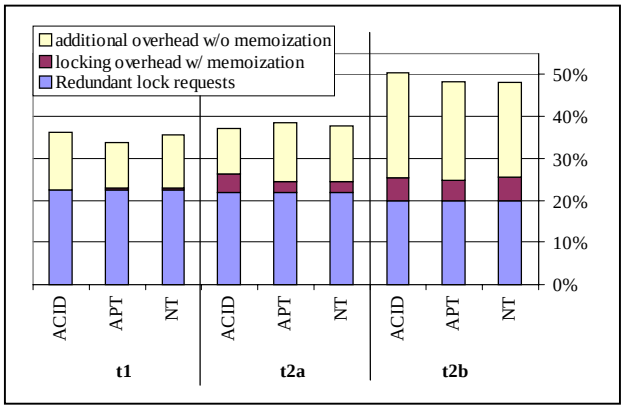


Figure 2: Overhead of isolation, no concurrency.

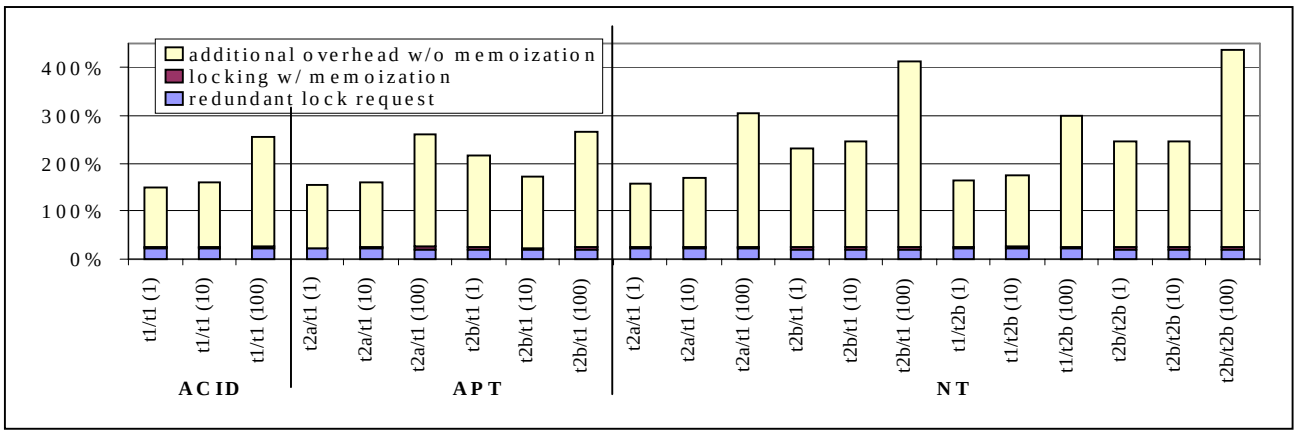


Figure 3: Processing overhead of isolation when shared data are accessed concurrently.

unlocked objects avoid looking up the TLS. Instead, the pointer to the shared lock state that represents the value of a lock owned only by the requester is retrieved from its locking context. Thus, all the expensive steps of the normal processing of lock requests are bypassed.

As a result, the overheads vary little from one transaction model to another (<1% difference). In all cases, the overhead of isolation ranges between 23% and 28%. Redundant lock requests contribute to between 80% and 98% of this overhead. Memoization, which in this case avoids trapping to the JVM runtime to forward lock requests to the lock manager, removes between 11% to 26% of overhead with respect to HSVM.

Figure 3 shows the overhead when various forms of concurrent sharing occur. Measurements were obtained by first running a number of transactions against the same working set so as to achieve 100% overlap, suspending them just before commit, and finally running the measured transaction to completion. As before, measurements are reported as overhead with respect to HSVM. Figure 3 uses ti/tj (n) to denote a scenario where a transaction performs an operation ti concurrently to n other transactions, each of which performs tj, where n can be 1, 10 or 100. ACID allows only read-only sharing, (i.e., only t1/t1 scenario can happen). APT allows read-only, read-write and write-read sharing. NT allows write-write sharing in addition to all previous forms of sharing.

We observed the same behavior as before, namely: the overhead of isolation remains between 22% and 28%, and redundant lock requests amount to more than 80% of it, regardless of the transaction model used and the features of the flexible isolation engine that they exploit. This result depends very much on memoization, which this time has a more subtantial impact on performance: disabling memoization results in execution times between 2 to 5 times slower than with HSVM, depending on the number of transaction operating concurrently over the shared data. Another important effect of memoization is to erase the differences that may result from different programming of conflict detection.

5.2 Memory footprint

Single address space approaches to multitasking seek to reduce the overall memory footprint of programs by sharing the runtime representation of classes, namely, the bytecodes, meta-data describing fields, methods, symbolic links, etc., and the dynamically compiled code produced for frequently used methods [CD01]. Transaction-based protection goes one step further by allowing application data to be shared as well. However, the space overhead intrinsic to the use of transactions can dilute these gains.

Lock barriers, which consist of a lock ownership test, and a memoized lock acquisition, increase the size of dynamically compiled code. For SPECjvm98 benchmarks this increase ranges from 28.6% (db) to 56.7% (mtrt); for OO7 the increase is 40%. This increase is penalizing when a compiled method is not actually shared because used by one program only.

The locking and logging used to enforce isolation also add to the overall runtime footprint of a program, although only under certain conditions: (i) when the program updates shared data, or/and (ii) when it manipulates shared data concurrently with other running programs. In all other cases, isolation adds to the footprint of a program 1 kB per locking context its transaction uses.

When programs access data concurrently, the space overhead of locking may offset the benefits of sharing data. Lock state sharing makes the space overhead of locking proportional to the number of distinct values of acquired locks, and not a function of the number of acquired locks. How many different lock values there are at a given time depends on the number of concurrent transactions and on how they overlap their working sets. This property allows locking to scale well with the number of locks needed, regardless of the granularity of locking. For instance, locking incurs exactly the same space overhead whether transactions traverse a small or a medium OO7 database, although traversing the latter acquires 9 times more locks. Figure 4 shows how the lock manager footprint evolves with the number of concurrent

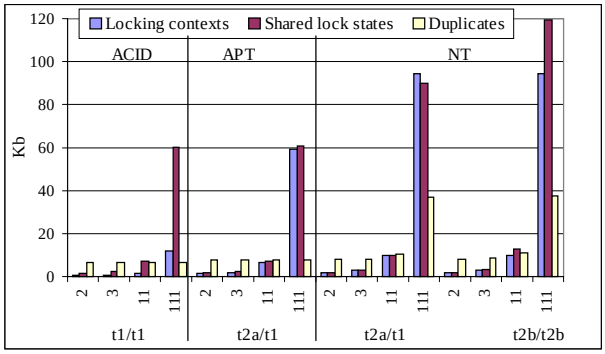


Figure 4: Lock manager footprint.

transactions. The size of bitmaps implementing set of locking contexts is 512 bits. Transaction models that use ignore-conflict relationships (e.g., APT and NT) consume more space to store them. Overall, the space overhead for locking remains small, between 1 to 6 kB per transaction.

6. Related Work

Applying transactions beyond the domain of databases is a recurring research theme. OS designers in the 80s embraced the transaction concept and many attempts to incorporate it into prototype distributed OSes are recorded in the literature. Two approaches were followed: those that use transactions internally [MMP83], and those that expose the transaction concept to programmers, either via toolkit or a programming language [SW91]. The nested transaction model was often chosen because of its ability to cope with failures without aborting a whole distributed computation. The granularity of concurrency control was typically coarse (e.g., a page).

Distributed programming language designers focused on integrating transactions with a language and its runtime. The Argus system, whose execution model relies on the nested transaction model, is a prominent effort in this direction [LCJ+87]. However, the performance reported hardly gives any idea of the overhead imposed by transactional mechanisms on the programming language runtime, and the technology gap between now and then makes comparisons with our system difficult. [LCJ+87] reports read lock acquisitions 10 times slower than another 14-instruction operation and redundant lock requests at half the costs of a lock acquisition. By comparison, our system processes the vast majority of lock acquisitions with 13 RISC instructions, and redundant lock requests with 7.

Recent research on extensible OSes has renewed the interest in transactional mechanisms, this time to shield the kernel against misbehaving extensions. The VINO system [SES+96] exemplifies this approach. Its kernel supports a limited form of nested transactions. Every interaction of an extension with the kernel is subject to transactional control. Each kernel function exposed to extensions includes a corresponding carefully crafted undo operation. As in our system, the transactional mechanisms are optimized to operate on volatile state. In contrast to VINO, our approach aims at protecting arbitrary user-defined programs from one another via transactional mechanisms automatically enforced by a programming language's runtime.

Our work also relates to research in designing flexible transaction processing engines. A landmark in this area is ACTA [Chry90], a comprehensive framework to formally specify and reason about the properties of extended transaction models. ACTA led to coining the notions of delegation and ignore-conflict relationships, which were integrated into the design of various flexible transaction processing engines [BDG+94, BP95, DAV97]. The lock managers of these systems, as well as all lock managers with support for nested transactions that we are aware of, extend textbook implementation of lock management [GR93], and as such are variations of the seminal design of System R*. Section 3.5 already discusses problems exhibited when this traditional design is incorporated into the runtime of a language-based extensible system. Our solution extends our previous work on lock state sharing to address these problems [DC01].

Some of the issues we have been facing are close to those encountered when transactional features are applied over main-memory data, such as in main-memory resident database systems and PPLs. Similarly to [GL92] direct pointers to lock data structures are exploited. The use of hardware protection to efficiently automate locking and logging was rejected because (i) the granularity of locking is too coarse, (ii) relocation of objects in locked pages is prohibited for the duration of the lock, and (iii) applications must either execute in separate address spaces or be able to restore, upon a thread context switch, the virtual memory protection corresponding to the state of a specific transaction.

Our support for undo differs from the noting of updates devised in the context of PPLs [HBM93], which typically employs remembered sets or card marking (some use hardware protection) to defer the generation of log records at transaction commit. In our case, recording of undo information cannot be deferred, since otherwise the old version of the object would be irrevocably lost.

7. Conclusions

Using transactions as protection domains in a language-based extensible system is appealing: it combines fault containment, safe termination, access control, and safe concurrency in a single construct. To explore this approach, we have extended a Java virtual machine with a flexible isolation engine that incorporates mechanisms, such as ignore-conflict relationships and delegation, to program advanced concurrency control. These features are supported by a lock management method that combines low-space overhead, high-performance, and programmable behavior. The resulting platform has been

used to realize three transaction models and evaluate their impact on program performance.

Our experiments show that, regardless of the transaction model used and of the sophistication of its locking rules, the overhead of isolation ranges between 22% and 28%. The use of memoization is key to limiting the processing overhead of isolation and to erasing the differences that would otherwise exists between various programming of the flexible isolation engine.

Finally, because at least 80% of the reported overhead is solely due to redundant lock requests, substantial improvements can be expected with the addition of more elaborate compiler optimizations (e.g., escape analysis), thus strengthening the case for transaction-based protection in language-based extensible system.

Trademarks

Acknowledgements

We would like to thank Mick Jordan for his comments on the various drafts of this paper, and Adam Welc who helped implementing the flexible lock manager during his summer internship at Sun Microsystems Laboratories.

References

[Anfi97] Anfindsen, O. Apotram – an application oriented transaction model. PhD thesis, Department of Informatics, University of Oslo, Norway. Research Report 215. 1997.

[BDG+94] Biliris, A., Dar, S., Gehani, N., Jagadish, H., and Ramamritham, K. ASSET: A System Supporting Extended Transactions. ACM SIGMOD Minneapolis, MN, 1994.

[BHL00] Back, G., Hsieh, W., and Lepreau, J. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. 4th OSDI, San Diego, CA, 2000.

[BP97] Barga, R and Pu C. A Practical and Modular Method to Implement Extended Transaction Models. 21st VLDB. Zurich, Switzerland, 1995.

[BR00] Bryce, C., and Razafimahefa C. An Approach to Safe Object Sharing. 15th ACM OOPSLA, Minneapolis, MN, 2000.

[BSP+95] Bershad, B., Savage, S., Pardyak, P., Becker, D., Fiuczynski, M., Sirer, E.G. Protection is a Software Issue. 5th Hot-OS, Orcas Island, WA, 1995.

[CD01] Czajkowski, G., Daynès, L. Multitasking without Compromise: a Virtual Machine Evolution. 16th ACM OOPSLA, Tampa, FL, 2001.

[CDN93] Carey, M., DeWitt, D., and Naughton, J. The OO7 Benchmark. ACM SIGMOD, Washington D.C., 1993.

[Chry90] Chrysanthis, P. ACTA: a framework for specifying and reasoning about transaction structure and behavior. ACM SIGMOD, Atlantic City, NJ, 1990.

[DAV97] Daynès, L., Atkinson, M., and Valduriez, P. Customizable Concurrency Control for Persistent Java. In Advanced Transaction Model and Architectures, Kluwer 1997.

[DC01] Daynès, L., and Czajkowski, G. High-Performance, Space-efficient, automated Object Locking. 17th IEEE ICDE, Heidelberg, Germany, 2001.

[DH99] Dieckmann, S., and Hözle, U. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. 13th ECOOP, Lisbon, Portugal,1999.

[GJS+00] Gosling, J., Joy, B., Steele, G. and Bracha, G. The Java Language Specification. 2nd Edition. Addison-Wesley, 2000.

[Gong99] Gong, Li. Inside Java 2 Platform Security. Addison Wesley, 1999.

[GR93] Gray, J., and Reuter, A. Transaction Processing: Concept and Techniques. Kluwer, 1993.

[HBM93] Hosking, A., Brown, E., and Moss, E. Update Logging for Persistent Programming Languages: A Comparative Performance Evaluation. 19th VLDB, Dublin, Ireland, 1993.

[HR93] Härder, T., and Rothermel, K. Concurrency Control Issues in Nested Transactions. VLDB Journal, 2(1), 1993.

[HCC+98] Hawblitzel, C., Chang, C-C., Czajkowski, G., Hu, D. and von Eicken, T. Implementing Multiple Protection Domains in Java. USENIX Annual Conference, New Orleans, LA, 1998.

[LCJ+87] Liskov, B., Curtis, D., Johnson, P. and Scheifler, R. Implementation of Argus. In 11th ACM SOSP. 1987,

[GL92] Gottemukala, V, and Lehman, T. Locking and Latching in main-memory resident database Systems. 18th VLDB, 1992.

[LY99] Lindholm, T., and Yellin, F. The Java Virtual Machine Specification. 2nd Ed. Addison-Wesley, 1999.

[Mich68] Michie, D. Memo' Function and Machine Learning. Nature, (218). 1968.

[MMP83] Mueller, E., Moore, J., and Popek, G. A Nested Transaction Mechanisms for LOCUS. 9th ACM SOSP, Bretton Woods, NH, 1983.

[RCW01] Rudys, A., Clements, J., and Wallach, D. Termination in Language-based Systems. Network and Distributed Systems Security, San Diego, CA, 2001.

[RSC92] Richardson, J., Schwarz, P., and Cabrera, L. CACL: Efficient Fine-Grained Protection for Objects. 7th ACM OOPSLA, Vancouver, BC, Canada, 1992.

[SES+96] Seltzer, M., Endo, Y., Small, C., and Smith, K. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. 2nd OSDI, Seattle, WA, 1996.

[Spec98] Standard Performance Evaluation Corporation. SPEC Java Virtual Machine Benchmark Suite. August 1998. http://www.spec.org/osg/jvm98.

[Sun00] Sun Microsystems, Inc. Java HotSpot™ Technology. http://java.sun.com/products/hotspot.

[SW91] Schmuck, F., and Wyllie, J. Experience with Transactions in Quicksilver. 13th ACM SOSP, Pacific Grove, CA, 1991.