Transactional Rollback for Language-Based Systems

Algis Rudys *arudys@cs.rice.edu*
Dan S. Wallach *dwallach@cs.rice.edu*
Department of Computer Science, Rice University

Abstract

Language run-time systems are routinely used to host potentially buggy or malicious codelets — software modules, agents, applets, etc. — in a secure environment. A number of techniques exist for managing access control to system services and even for terminating codelets once they've been determined to be misbehaving. However, because codelets can be terminated anywhere in their execution, a codelet's internal state might become inconsistent; restarting the codelet could result in unexpected behavior. Any state the codelet shares with other codelets may likewise become inconsistent, destabilizing those codelets as well. To address these problems, we have designed a mechanism, strictly using code-to-code transformations, which provides transactional rollback support for codelets. Each instance of a codelet is run in its own transaction, and standard (ACID) transactional semantics apply. All changes made by the codelet are automatically rolled back when the corresponding transaction aborts. We discuss a transactional rollback implementation for Java, and present its performance.

1. Introduction

Language run-time systems are routinely used to host potentially buggy or malicious *codelets*[1] in a secure environment. Maintaining such secure environment requires some control over the execution of these codelets. At its most basic level, this means the ability to mediate between a codelet and the potentially dangerous primitives it is allowed to call, as well as the ability to stop and start codelets at will.

A number of mechanisms exist for access control and for safe termination of codelets. However, language run-time systems suffer the dual problem of restarting terminated codelets. Language run-time systems allow and encourage unrestricted data sharing among codelets; these shared data structures could be in an inconsistent state when the codelet is terminated. These inconsistencies can destabilize other running codelets as well as make it complicated to restart a terminated codelet.

One approach to addressing this problem is equivalent to an approach taken for code termination: if data sharing is the problem, simply disallow data sharing. If all the state a codelet ever touches is destroyed with the codelet, there is no restart problem. However, this severely restricts the capabilities of the languages. It also means programs written for the original language system, which could depend on data sharing, might need to be rewritten to work in this new environment.

We propose a language-based solution to address this problem. We observe that many codelet systems, such as component-based web servers (*e.g.* a Java Servlet web server), execute in a transactional style, spawning codelet instances to independently service requests as they arrive. We track the changes the codelets make to any state; if a codelet is terminated, we roll back all of its changes. If the codelet is later restarted, the system is not in some intermediate state that might cause instability. Because of the nature of the rollback operation, codelets must be run as transactions in order to maintain the consistency of the state. Note, however, that we are not saving the codelet's call stack or execution state — that is, we are not saving a continuation. When the codelet is resumed, this state is initialized from scratch. Only the codelet's persistent state is saved. Similarly, transactional rollback only deals with operations on memory, and doesn't address network- or file-based state. From the perspective of writing codelets, our design only requires the addition of transaction *start* and *commit* instructions to the system that hosts codelets. We expect these transactional instructions would be integrated in the event dispatching mechanism used in such systems as Web server plug-ins. No other changes would need to be visible to the codelet programmer.

In this paper, we present the design and implementation of a reusable framework for introducing transactional

[1] The term "codelet" is also used in artificial intelligence, numerical processing, XML tag processing, and PDA software, all with slightly different meanings. When we say "codelet," we refer to a small program meant to be executed in conjunction with or as an internal component of a larger program.

rollback to programs at the language level. Section 2 discusses our design goals and presents and justifies our design. In Section 3, we discuss a Java-based implementation of our transactional rollback design, and address a number of implementation-specific issues. Section 4 presents performance results.

2. System design

The ability to terminate codelets, assumed to be executing transiently in an otherwise long-running system, is a necessary property of this long-running system. We have addressed this problem with soft termination [31]. However, as a result of a codelet's untimely termination, shared state might become inconsistent. To address this, we need a mechanism to return the system to a known-consistent state.

A number of possible designs exist for rolling back state in a language run-time system. We begin by explaining how transactions are an appropriate mechanism for rollback. We then explore the range of possible designs, and explain how we chose to approach the problem. We also address issues in the design of rollback.

2.1. Rollback and transactions

The most straightforward solution for designing language-based rollback is to simply keep a record of changes made by a codelet, and roll the changes back if the codelet is terminated. However, due to the concurrent nature of multi-threaded language systems, where multiple codelets may read and write to shared data in parallel, termination could still result in inconsistent state. Two general types of data conflicts complicate our design, read-after-write conflicts and write-after-write conflicts.

Write-after-write conflicts occur when two codelets write to the same variable. If one of the codelets is terminated, it becomes unclear how to roll back that transaction's writes to the variable. Read-after-write conflicts occur when one codelet reads a variable another codelet has previously written to. If the writing codelet is terminated, the value read by the reading codelet becomes invalid, and the reading codelet must now be terminated, or at least restarted, as well.

Write-after-write and read-after-write conflicts are well known in the domain of databases. They are generally addressed by ensuring that all state is operated on strictly by ACID (atomic, consistent, isolated, durable) transactions. In particular, we can prevent these conflicts by constraining the order of operations within transactions to prevent the offending cases. The resulting order is said to have the properties of *serializability* and *strictness*. Conveniently, a well known locking protocol, *strict two-phase locking*, guarantees these properties. Any textbook on databases, such as Silbershatz *et al.* [33], provides a more in-depth coverage of this material.

For our system, then, each codelet runs within a transaction, utilizing the system memory as a database. In the rest of this section, we show how this can be integrated into a language system. We also discuss some issues that arise in adding database functionality to a language run-time system for supporting transactional rollback.

2.2. Architecture of transactional rollback

Transactional rollback has a number of similarities in design to language persistence. Persistence is the notion that the state of a program is maintained even in cases as extreme as the computer rebooting. To accomplish this, the system must keep track of which data a program accesses. At prescribed points in the code, the system must save the state to disk, making it persistent. It must also deal with system failures before the data has been written to disk. Similarly, transactional rollback must keep track of which state a codelet has modified. If the codelet is terminated, the changes the codelet has made must be rolled back to stored, stable values. The system might track this meta-state at many different levels.

Likewise, there are numerous parallels between transactional rollback and language security. In language security, the basic goal is to protect system-level invariants, such as codelet separation and resource limitations. Transactional rollback is concerned with protecting unspecified invariants at the user level.

Like language persistence and language security, we can design transactional rollback to run below the language run-time system, inside the run-time system, or above the run-time system.

2.2.1. Below the run-time system. At the lowest level, transactional rollback can be designed to operate below the language run-time system, generally as a service of the operating system. Since operating system mechanisms are simple and well-understood, greater assurance can be provided by adding transactional behavior at this level. However, the operating system only sees pages and words in memory, and does not understand the semantics of the data it is operating on. As a result it cannot take advantage of these semantics to prevent unnecessary contention for system resources.

Numerous persistent object systems operate at the operating system level. The operating system can use the page access patterns of running codelets to determine which pages of memory need to be made persistent. This is the approach taken by such persistent systems as Grasshopper [12], KeyKOS [15], and others. Similar approaches are taken by software distributed shared memory systems [30] to propagate changes. Finally, Howell [19] describes an implementation of Java persistence which operates above the operating system but below the language runtime. A more complete discussion of operating system support for persistence and transactional systems is provided by Dearle and Hulse [11].

The operating system can also be used to enforce language security invariants. For language security, the operating system already has built-in protections for cross-domain access of state. However, the operating system uses processes to separate protection domains, so individual codelets have to be run in separate JVM processes. At a more extreme level, one can take advantage of the separation afforded by running the codelets on different machines entirely. Several systems use these mechanisms to provide language security [24, 34].

2.2.2. Inside the run-time system. We can also implement transactional rollback as a customization to the language run-time system. With the language run-time system's semantic understanding of the language's data structures, we can provide transactional rollback at the granularity of these data structures. The language run-time system implementation itself does not necessarily suffer from the performance or design constraints imposed on codelets running above the language run-time system. However, this approach suffers from a lack of portability; to provide transactional rollback in different implementations of the same programming language, the design must be re-implemented for each language run-time system.

Language persistence is also commonly integrated into the language run-time. The language run-time system understands the data structures a program is using, and can thus save and restore data at the granularity of objects, rather than individual memory words or large data pages. This more precise granularity can result in fewer cases of false sharing contention (that is, cases where two codelets are accessing memory that is within the same memory page, but is actually used for separate and unrelated objects). Most persistent language-based systems track changes at the granularity of objects. This granularity has also been shown to reduce contention in distributed shared memory systems [20].

This approach is the earliest in persistent systems. Persistent systems grew up around such early persistent programming languages as PS-Algol [2] and Elle [1], as well as the later Napier88 [10], in which persistence support existed as a necessary part of the language run-time system. The earliest persistent object system was POMS [7]; more recent examples include Thor [23] and Mneme [26]. Java has been a specific target of modifications to support persistence, with such systems as PJama [29].

Finally, the language run-time system can provide enforcement of language security. The language run-time system understands the data structures a program is using, and can use this information to provide more precise protection. In fact, many systems enforcing language security exist as part of the language run-time systems because these aspects of language security are actually integrated into the design of the language. A number of Java resource management systems also rely on customizations to the JVM [3, 4, 5, 35]. Some, such as PERC [27], even go so far as to modify the language to provide certain resource guarantees. Vino [32] directly tracks the resources used by its codelets, allowing for a limited form of transactional rollback.

2.2.3. Above the run-time system. Designing transactional rollback to run on top of the language run-time system solves the issue of portability. Since the transactional rollback system is designed in terms of the language itself, any implementation of the language run-time system can use the transactional rollback system unmodified. What's more, code-to-code transformations are well-understood in language theory, and a high-level design based on code-to-code transformations could be more easily adapted to work with many different programming languages. However, these code-to-code transformation systems suffer performance and flexibility penalties because they are unable to modify all aspects of the underlying language run-time system. In our previous work on soft termination [31], facing these same tradeoffs, we chose to implement our system as a code-to-code transformation, and the performance penalties were still quite reasonable (worst case benchmarks, doing numerical processing, experienced an 18-25% overhead, where other benchmarks experienced only a 3-6% overhead).

This approach has been used in designing language persistence. In such systems, the codelet is transformed at runtime to provide the persistent system, also running in the language run-time system, with access information for that codelet. Marquez *et al.* [25] describe a persistent system implemented in Java entirely using bytecode transformations at class load time.

Some aspects of language security can also be enforced on top of the language run-time system. In these systems, the codelet is transformed as it is loaded by the run-time system to make run-time checks enforcing language invariants. These run-time checks likewise run entirely on top of the language run-time system. Such checks have been used in the past for access control [14, 28, 36], resource management [6, 8, 16, 22], and to allow for program termination [31].

2.3. Design discussion

Because we want a portable system, we have chosen to implement transactional rollback above the language run-time system using code-to-code transformations, as described in section 2.2.3 above. We begin my describing the transactional rollback transformation. We also describe some complications inherent in designing transactional rollback.

2.3.1. The transactional rollback transformation. The code-to-code transformation for transactional rollback is relatively straightforward. First, each subroutine of the codelet is duplicated, and a parameter is added to the duplicate subroutines. This parameter is an object representing the current transaction, and is used to identify the current codelet. In addition, all subroutine applications from duplicate subroutines are rewritten to instead call the duplicate-equivalent subroutines, passing the current transaction parameter along. This effectively duplicates the call graph of the entire system, forming a transactional and a non-transactional context for each codelet.

Each object instance is mapped one-to-one to a lock. The duplicate subroutines are rewritten such that before a codelet can access a data structure, the transaction which corresponds to the codelet must first acquire the lock corresponding to the data structure. As mentioned in Section 2.1 above, lock acquisition follows the strict two-phase locking protocol to ensure the system's consistency. If a deadlock is detected, the codelet is terminated, and its modifications rolled back. The original subroutines remain unmodified, allowing for minimal overhead in the event that a non-transactional context is desired; the cost of this flexibility is a $2\times$ overhead in the code size of the program.

When a write lock is granted, the corresponding data structure must additionally be backed up. A reference is thus maintained from every data structure to its shadow backup. If the current transaction is aborted, any modifications will be rolled back, thus additionally requiring the maintenance of references from the backups to their original locations. When the write lock is granted, a shallow copy of the data structure is made into the backup. Deep copies are unnecessary, as any data structures referred to by the current data structure are themselves backed up when write-locked. Note that a data structure is only backed up once per transaction, the first time the write lock is acquired. Figures 1–3 illustrate a transaction reading from and writing to an object.

If a transaction commits, any backups being maintained for that transaction's write locks are thrown away. If the transaction aborts, its backups are restored. Aborts occur whenever the corresponding codelet abnormally terminates, either because of a bug, in response to a termination request by the system, or to break a deadlock. Locks are released when the transaction finishes, either by committing or aborting.

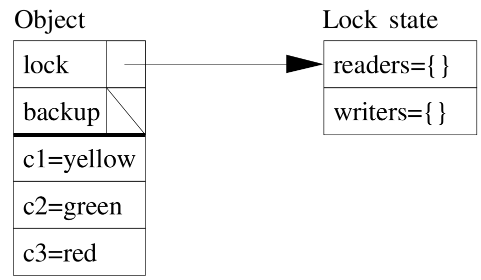

Figure 1. Before a transaction has begun operating on the object, its backup is empty and it points to an empty lock. The original object consisted only of the fields *c1*, *c2*, and *c3*. The *backup* and *lock* fields were added in transforming the object for transactional rollback.

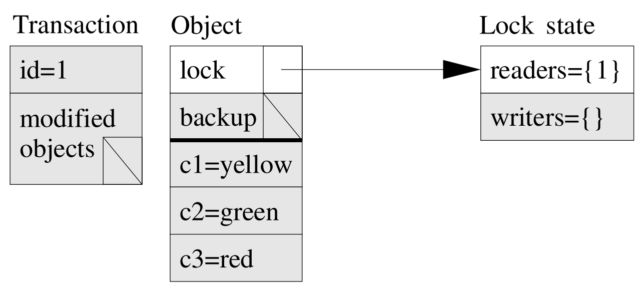

Figure 2. Transaction 1 has read from the object. It has a read lock on the object, but the object has not been written to, so no backup has been created. Grayed-out fields are those which are not modified in the operation.

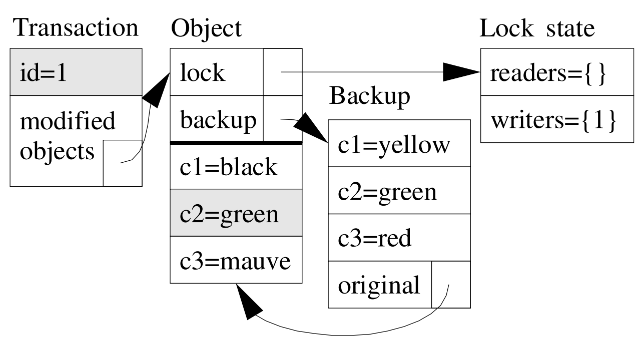

Figure 3. Transaction 1 has written to the object. It has a write lock on the object, and the backup has been allocated and filled with the old values of each field. Grayed-out fields are those which are not modified in the operation.

2.3.2. Deadlocks. Introducing transactions to a language, as in transactional rollback, can cause a previously deadlock-free program to deadlock. These deadlocks must be dealt with, either by preventing them beforehand or by detecting them after the fact. Deadlock prevention algorithms involve acquiring locks before they are needed and in a well-defined order. Unfortunately, codelets are not written to use any particular lock access discipline (and are not, in general, designed around the thought that they might be running in a transactional environment), and attempting to statically add such a discipline to a codelet would be infeasible without grossly overestimating the lock usage of a codelet. This makes it unreasonable to attempt an implementation of deadlock prevention.

Deadlock detection simply involves maintaining a directed graph of lock dependencies. If this graph ever develops a cycle, there is a deadlock. This can be determined by performing a simple graph traversal when a lock is requested. If a cycle is detected, the newest transaction in the cycle is terminated. This guarantees that at least one transaction in the system (the oldest) will never be terminated due to deadlock, and therefore that deadlock will never prevent the system from making forward progress. These concepts are well-known in the field of databases [33].

Note that locks available to a language are not directly used in transactional rollback, although such language-based locks, if available, may be used short-term within critical sections of the lock manager. As a result, our deadlock-detection scheme does not address deadlocks which are preexisting in the codelet. This also introduces the possibility that codelets may deadlock on a mix of normal language-based locks and transactional rollback locks.

If the lock-state of language-based locks is accessible to our deadlock-detection algorithm, we can incorporate this information to detect such "mixed" deadlocks. Otherwise, we could implement a timeout for transactional rollback locks. When some application-specific heuristic function determines that no forward progress is occurring (for instance, by detecting that no lock activity has occurred for one minute), we can begin aborting transactions until the system resumes making forward progress.

3. Implementation

We chose to implement transactional rollback for the Java programming language, giving us experience that directly applies to a popular language and which can be easily applied to other object-oriented languages. We also favored Java for the presence of preexisting tools to help implement and debug the code-to-code transformations.

In this section, we discuss how to transform Java code to support transactional rollback. We then discuss a number of implementation issues and how we addressed them. Some are Java-specific, and would not pose a problem for other language systems, while others are universal.

Our implementation of transactional rollback utilizes Java bytecode rewriting; we use IBM's JikesBT[2] bytecode toolkit.

3.1. Locking code insertion

The systems we have in mind in our design of transactional rollback are those which run potentially untrusted code. As a result, we make no assumptions about the code as it is input into the system, and the system itself must ensure that the code has been transformed to support transactional rollback. We can ensure this by performing the transactional rollback transformations in the class loader, as the class is being loaded into the JVM.

A compiled Java program is represented as a set of `.class` files, each of which contains the bytecode for a single Java class. These classes are loaded, typically on-demand, by a *class loader*. Class loaders are built into most JVMs and can be extended to support other functionality, such as rewriting code as it is loaded. By performing code transformations in the class loader, the point through which all code is loaded into the system, we can guarantee that all code in the system will be consistently transformed.

The transformer implements the design described in Section 2.3 for transactional rollback. All methods are duplicated, and a `Transaction` parameter is added to the end of the parameter list of the duplicate methods. All method calls within these duplicate methods are then rewritten to call the duplicates of the original targets, passing along the `Transaction` argument from the caller to the callee. Finally, all state accesses in these duplicate methods are preceded by calls to the lock manager.

A number of instance fields need to be added to every class. First, every instance must have a reference to a lock object. Since we employ lock state sharing (see Section 3.2, below), any state specific to the object (such as other transactions waiting for the object to become free) need also be stored in the object. Finally, we need a way to store backup data for when a codelet must be rolled back. To address this, two new Java classes are created. The *instance backup class* stores backups for instances variables; it is instantiated on demand by the backup routine. The *static backup class* stores backups for static class variables; it is instantiated when the class is initialized.

The backup operation, when called on an object, saves the fields of that object into the respective backup object, instantiating the backup object if necessary. The fields are copied using Java's assignment operator. If the field is a primitive, then rollback will restore the appropriate value. If the field is an object, then any modifications of that object

[2] `http://www.alphaworks.ibm.com/tech/jikesbt`

require write locks on that object. If the backup operation is working on an array, as opposed to an object, similar backups are made, but some special handling is necessary. See Section 3.3 for details.

The static backup class services an additional purpose. When a class has been rewritten, it is also declared to implement the `TransObject` interface. This interface allows the transactional system to operate on transaction-enabled objects in a uniform manner. As static classes cannot be passed as parameters and cannot be operated on through an interface, we instead use the static backup class. It is created as an implementation of `TransObject`, wrapping the static class for the transaction system.

3.2. Lock state sharing

We implement two-phase locking at the granularity of object instances to guarantee consistency. We used object instance granularity as a matter of convenience. In particular, it allows us to use Java interfaces as the primary interface between the lock manager and the units of locking. We can also store some lock state in objects, eliminating the overhead of separately allocating additional objects.

Note that the lock manager must maintain lock state for each object in the system. There will tend to be many more objects in a system than active transactions operating on these objects, and multiple objects will tend to have the same lock state. Traditionally, a lock object would need to be allocated and instantiated for each object that was created. To take advantage of this pattern, we use NLSS [9], a form of *lock state sharing*. In such a system, objects with the same lock state (that is, objects locked by the same transactions and in the same modes) will have references to the same lock object. NLSS additionally eliminates the need to maintain mappings from each transaction to the objects that the transaction has locked for reading. We still must maintain a mapping from each transaction to the objects it has write-locked, for the purpose of rollback. We expect this set to be much smaller, however.

3.3. Arrays

Arrays are treated specially in Java. On the one hand, they are instances of `java.lang.Object`; they cannot be treated as primitives, because they are composite data structures. In addition, two separate objects could maintain references to the same array; it is necessary to maintain lock state for each array independently. On the other hand, it is not possible to add or change methods or fields of arrays, so they cannot be treated like other objects.

Since we must maintain the lock state for each individual array and we cannot modify the array implementation, we must maintain an external mapping from arrays to their lock state. A hash table is used to store this mapping. To lower the cost of hash table accesses, we memoize hash table queries. In the common case, when the lock state we are looking for is in the memo table, the additional work performed when locking an array versus a regular object is a method call to get the array's `hashCode()` and a method call to retrieve the lock from the memo table. In practice, codelets which rely heavily on array manipulation will experience significant slowdowns in our system. Fixing this would require access to the Java implementation for arrays to directly add a reference to the array's lock in the array object's header.

3.4. Constructors

Constructor methods also need to be dealt with specially in Java. Arbitrary work can be done in constructors, so they must be rewritten just like any other Java method. Java constructors are also responsible for initializing variables to appropriate default values. This includes the new fields which are introduced as part of the transactional rollback (*e.g.*, the lock field, the backup fields).

We observe an opportunity for optimization here, as noted in Daynès and Czajkowski [9]. When a codelet running inside a transaction instantiates a new object, it is only visible to that codelet until either the transaction aborts, and the object becomes unreachable; or the transaction commits. Additionally, the very act of instantiation writes to the object.

Therefore, when an object is instantiated, we initialize the lock pointer of the object to the *single write owner* for the transaction. The single write owner is a lock structure which identifies the transaction as having an exclusive write lock. Since lock states are shared, we only need to create the single write owner once for each transaction. This optimization eliminates the need for newly instantiated objects to be explicitly (and inevitably) locked, reducing overhead, particularly if we implement a fast-path, as discussed in Section 3.7 below.

3.5. Native methods

Java programs can invoke *native methods* (methods not written in Java), which cannot be transformed; that is, we cannot pass `Transaction` objects to the native methods. This is particularly problematic because the native code might then call back to Java code, and needs some way of getting the current transaction. To solve this problem, we rewrite Java classes to store the current transaction with the current thread before calling a native method. When control returns from a native method, we can restore the current transaction. Our current system only adds the transaction restoration logic at up-call points that we know occur in our benchmark tests. Doing this more generally would require similar analysis and transformations to that performed in

SAFKASI [36]. SAFKASI showed performance overheads of 15-30% for passing its security context argument to all methods in the system. We would expect a similar overhead here.

3.6. Open files

As noted by Howell [19] and others, open files and network connections pose a problem because they can be used to store state where the transactional system cannot roll it back. They also themselves represent state (such as file offsets) that is managed natively by the language run-time system or operating system. Should the codelet terminate prematurely, there is no way for the transactional system to roll back any writes to files or onto the network, or otherwise restore the state of the object.

We observe, however, that multiple instances of a codelet will not tend to share the same open file descriptor or network socket. As a result, if a codelet terminates prematurely, all of its open file descriptors and network sockets, which were created while the codelet was running, become inaccessible. If multiple instances of a codelet share an open file in an environment where codelets can fail asynchronously, the system is equally at risk of inconsistent data in files with or without transactional rollback; we do not exacerbate the problem.

3.7. Optimizations

We implemented a number of optimizations on the code, and can foresee a number of others that may speed the code up considerably. The optimizations we did implement were inspired by the results of profiling runs of the code. This is discussed in Section 4.3. The simplest optimization we use is a fast path. As part of the thread locking transformation, we insert code inline to check whether the locking operation is necessary. Only if we determine that a lock needs to be acquired do we actually attempt to acquire it. This will generally save at least one method call per field access, and could potentially save many more.

As a more ambitious optimization, we observed that the most commonly accessed object in a method is `this`, the object upon which instance methods generally operate. In almost all cases, a method will access its `this` object. As a result, if we can statically determine that a method accesses the `this` object at least once, we can acquire a single lock for that object at the beginning of the method. If any accesses are writes, we acquire a write lock. Otherwise, we acquire a read lock. All subsequent locks of `this` are omitted.

We would like to extend the above to apply to all objects in a method. If we could statically determine that multiple locks are acquired for the same object, all but the first could be removed. Similarly, if a read lock request and a write lock request were made on the same object from the same method, the read lock request would be redundant.

To address this problem, we could use a variation of lazy code motion [21, 13], a well known compiler optimization. Using this technique, we could eliminate more of the redundant lock acquisitions within a method. Since this method has been shown to be optimal, any redundant lock requests that can be detected will be detected with this method. Lazy code motion also guarantees that no code paths are extended, meaning a lock request is only made if it is necessary. Hosking *et al.* [17] discuss similar methods for optimizing such a system. Note that with any of these methods, there still needs to be one lock acquisition per object per method. Inter-procedural analysis may help further eliminate these lock operations.

3.8. Integration into a real-world system

As mentioned in Section 2, we assume that this system is already running in a pseudo-transactional environment. That is, there is a long-running system with a number of transient codelets running concurrently in separate threads and providing some services to the system. In this case, starting a transaction can coincides with starting a codelet thread: the method `Transaction.doTransaction()` is called with a `java.lang.Runnable` object as a parameter. The `run()` method of this Runnable object, as rewritten by the transactional rollback transformer, is the entry point to the transaction. When the method returns, either normally or abnormally, the transaction is completed, either having committed or aborted, respectively. If the transaction was aborted, an exception is thrown. The system hosting the codelets might choose to implement a loop which automatically restarts codelets if they are aborted.

4. Performance

We measured the performance of our system using an AMD Athlon workstation (1.2 GHz CPU with 512 MB of RAM, running Linux version 2.4.16) and version 1.3 of Sun's Java 2 JVM, which includes the HotSpot JIT.

We performed two classes of tests: microbenchmarks which measure the performance of specific Java language constructs, and some real-world programs.

We performed three different measurements for each benchmark. The first was the unmodified benchmark. In the second, the class files were rewritten, but were not executed in transactional contexts. This case was designed to strictly show the performance overhead of the larger class files and additional per-object data storage (with nothing stored there, but taking up more space). Finally, we ran each benchmark in a transaction to measure the total overhead of the transaction system. Finally, for the application

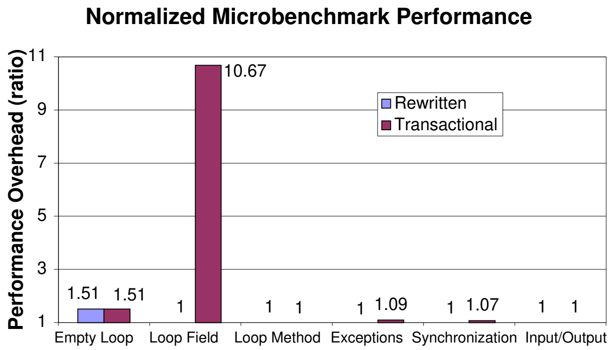

Figure 4. Performance of rewritten microbenchmark class files, run both inside and outside of a transactional context, relative to the performance of the corresponding original class files.

benchmarks, we further measured the performance with an "array cheat" which we will describe below.

4.1. Microbenchmarks

We first measured a series of microbenchmarks to stress-test the JVM with certain language constructs: looping, method and field accesses, exception handling, synchronization, and I/O. We used a microbenchmark package developed at University of California, San Diego, and modified at University of Arizona for the Sumatra Project[3]. The results are shown in Figure 4.

The microbenchmark results were entirely unsurprising. In the benchmark which focuses on field and array accesses, the overhead of the transactional system was extreme (over $10\times$ overhead). All other test benchmarks had significantly lower overhead. The only other microbenchmark with an overhead of more than 10% was the loop test, at $1.5\times$. However, this reflects an actual margin of 0.008 seconds, and the overhead shows up for the rewritten classes in both transactional and non-transactional contexts. Because it is so small in wall-clock time, we cannot conclusively determine the cause of this overhead.

[3]The original web site is `http://www-cse.ucsd.edu/users/wgg/JavaProf/javaprof.html`. The source we used was distributed from `http://www.cs.arizona.edu/sumatra/ftp/benchmarks/Benchmark.java`

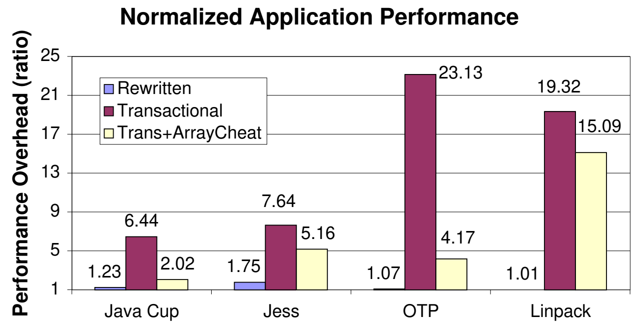

Figure 5. Performance of rewritten application class files, run both inside and outside of a transactional context, as well as inside with the array cheat enabled, relative to the performance of the corresponding original class files.

4.2. Application benchmarks

We benchmarked the real-world applications JavaCup[4], Linpack[5], Jess[6], and OTP[7], to get a feel for the performance impact of our transformations on real-world code. JavaCup is a LALR parser-generator for Java. Jess is an expert system shell. Linpack is a loop-intensive floating-point benchmark. OTP is a one-time password generator which uses a cryptographic hash function. The results are shown in Figure 5.

The first trend we noticed was in the overhead of the transformed code not running in a transactional context. Jess and JavaCup demonstrated significant overheads, whereas OTP and Linpack did not. In searching for a cause for this overhead, we considered growth in the heap size and growth in the class files. Unfortunately, no strong correlation appeared between measured increases in these values and the performance values we present in Figure 5. Java 1.3 does not provide very sophisticated heap profiling tools, so we cannot rule out the change in memory usage as the culprit. Further investigation would require more sophisticated profiling tools than we have at our disposal.

When transactions were enabled, the overheads were impressively large — performance overhead ranged from a factor of $6\times$ to $23\times$. Clearly, these numbers indicate our system is not yet suitable for deployment in practice. We suspected that a major component of this overhead was related to our handling of arrays (see Section 3.3). To study

[4]`http://www.cs.princeton.edu/~appel/modern/java/CUP/`

[5]`http://netlib2.cs.utk.edu/benchmark/linpackjava/`

[6]`http://herzberg1.ca.sandia.gov/jess/`

[7]`http://www.cs.umd.edu/~harry/jotp/`

this further, we implemented an "array cheat." While no longer semantically sound, this cheat eliminates the hash table lookups to find the per-array locks (one global lock is used), and it no longer performs backups of arrays. This cheat represents an upper bound on the performance benefit that might be achieved with a hypothetically extensible array implementation, giving us a slot per array to store the locks. Our measurements show significant gains relative to the original transactional system (reducing the overhead to between a factor of $2\times$ and $15\times$). Excluding Linpack, which is a fundamentally array-driven benchmark, the overhead experienced on the other application benchmarks with the array cheat was at most $5\times$. The relative speedup on OTP was quite impressive (from $23\times$ to $4\times$). OTP allocates a large number of small arrays which it uses for temporary storage and which then quickly become garbage. Clearly, such a program introduces a large overhead when external references must be kept to each temporary array.

An interesting question is how these performance numbers compare to other systems that attempt to solve similar problems. When reading the literature on persistent object systems, we have not found many papers willing to compare their performance to the original, non-persistent system. Marquez *et al.* [25] present a code-to-code transformation that implements orthogonal persistence and compare its performance to JDK 1.2 with no support for persistence. They indicate their system, with a warm disk cache, has a roughly $9\times$ performance cost relative to JDK 1.2 when running their test benchmark. This confirms that we are "in the ball park" but further optimization is still necessary.

4.3. Optimization profiling

In order to gain insight into the best methods for optimizing the system, we instrumented the transformed classes to provide an accounting of when locks were acquired. This includes attempts to acquire locks when the fast path determined the acquisition is not necessary. Our goal was to develop a strategy to allow the transformer to statically determine that a lock acquisition is not necessary and omit it.

In analyzing these results, an interesting pattern emerged: a large portion, and for some benchmarks, the vast majority, of redundant lock requests were on the `this` object. This result is because a method will access fields of its own object far more frequently than fields of other objects. We noted that it is never safe to remove all lock acquisitions for an object from a single function without performing inter-procedural analysis, but even considering this requisite acquisition, there were still many redundant lock requests of `this`.

Our solution, and the only optimization we performed for our system, was, whenever we could statically detect that a method would be accessing the `this` object, we acquire the appropriate lock to `this` at the beginning of the method and omit any lock requests for `this` anywhere else in the method. This reduces many redundant lock requests while adding limited deadlock pressure. The result is best illustrated in the OTP benchmark.

OTP uses the MD5 hash function to generate one-time passwords. The MD5 implementation keeps mathematical state in object fields, and performs long sequences of mathematical operations on them. Before this optimization, each mathematical operation needed to be prefixed by a lock acquisition. Afterwards, a lock acquisition was only needed at the beginning of each function. The result was a substantial drop in overhead for this benchmark (and smaller drops for other benchmarks).

Even with this optimization, however, there is still considerable room for improvement. There are still a large number of subsequent lock acquisitions to the same object, read lock acquisitions followed closely by write lock acquisitions, and other redundant lock acquisitions. As discussed in section 3.7, data flow and control flow analysis could be used to identify redundant lock requests and reduce this overhead.

5. Conclusion

Termination is a crucial capability for providing resource control in language-based systems. In the face of data sharing, the ability to roll the system back to a safe state and to safely restart programs becomes an equally important problem in such systems. Transactional rollback provides a language-based portable solution to the problem of restarting codelets. Our design is independent of any particular language run-time system, allowing implementations which do not depend on a single language. Our Java implementation shows a worst-case overhead of $23\times$, with overheads of 6-$7\times$ in the absence of extensive array usage. While these overheads are quite large relative to the original system's performance, they represent a starting point for semantics that are otherwise unavailable to the designer of systems that must reliably execute untrusted or buggy codelets. A number of opportunities exist to further optimize our system, particularly with regard to making small modifications to the run-time system to allow us to annotate arrays with direct references to their backup information. With small run-time modifications, we believe we can offer transactional rollback semantics for codelets with reasonable performance.

Acknowledgements

We would like to thank Willy Zwaenepoel and Dave Johnson for helpful discussions. We would also like to thank the anonymous DSN 2002 reviewers for their comments.

This work is supported by NSF Grant CCR-9985332.

References

[1] A. Albano, M. E. Occhiuto, and R. Orsini. A uniform management of temporary and persistent complex data in high level languages. Technical Report S80-15, Università di Pisa, Sept. 1980.

[2] M. Atkinson, K. Chisholm, and P. Cockshott. PS-Algol: an Algol with a persistent heap. *SIGPLAN Notices*, 17(7):24–31, July 1982.

[3] G. Back and W. Hsieh. Drawing the Red Line in Java. In *Proceedings of the Seventh IEEE Workshop on Hot Topics in Operating Systems*, Rio Rico, Arizona, Mar. 1999.

[4] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, California, Oct. 2000.

[5] P. Bernadat, D. Lambright, and F. Travostino. Towards a resource-safe Java for service guarantees in uncooperative environments. In *IEEE Workshop on Programming Languages for Real-Time Industrial Applications*, Madrid, Spain, Dec. 1998.

[6] A. Chander, J. C. Mitchell, and I. Shin. Mobile code security by Java bytecode instrumentation. In *2001 DARPA Information Survivability Conference & Exposition (DISCEX II)*, Anaheim, California, June 2001.

[7] W. P. Cockshott, M. P. Atkinson, K. J. Chisholm, P. J. Bailey, and R. Morrison. A persistent object management system. *Software - Practice and Experience*, 14(1):49–71, January 1984.

[8] G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 21–35, Vancouver, British Columbia, Oct. 1998.

[9] L. Daynès and G. Czajkowski. High-performance, space-efficient, automated object locking. In *Proceedings of the 17th International Conference on Data Engineering*, Heidelberg, Germany, Apr. 2001.

[10] A. Dearle, R. Connor, F. Brown, and R. Morrison. Napier88—a database programming language. In R. Hull, R. Morrison, and D. Stemple, editors, *Second International Workshop on Database Programming Languages*, pages 213–229, June 1989.

[11] A. Dearle and D. Hulse. Operating system support for persistent systems: Past, present and future. *Software: Practice and Experience Special Issue: Persistent Object Systems*, 30(4):295–324, 2000.

[12] A. Dearle, J. Rosenberg, F. Henskens, F. Vaughan, and K. Maciunas. An examination of operating system support for persistent object systems. In *Proceedings of the Twenty-Fifth Annual Hawaii International Conference on System Sciences*, pages 779–789, 1992.

[13] K.-H. Drechsler and M. P. Stadel. A variation of Knoop, Rüthing and Steffen's *lazy code motion*. *SIGPLAN Notices*, 28(5):29–38, May 1993.

[14] U. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 246–255, Berkeley, California, May 2000.

[15] W. S. Frantz and C. R. Landau. Object-oriented transaction processing in the KeyKOS microkernel. In *Proceedings of the Usenix Symposium on Microkernels and Other Kernel Architectures*, pages 13–26, San Diego, California, Sept. 1993.

[16] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *USENIX Annual Technical Conference*, New Orleans, Louisiana, June 1998. USENIX.

[17] A. Hosking, N. Nystrom, Q. Cutts, and K. Brahnmath. Optimizing the read and write barrier for orthogonal persistence. In *Eighth International Workshop on Persistent Object Systems: Design, Implementation and Use*, Tiburon, California, Aug. 1998.

[18] A. L. Hosking and J. E. B. Moss. Lightweight write detection and checkpointing for fine-grained persistence. Technical Report 95-084, Department of Computer Sciences, Purdue University, Dec. 1995.

[19] J. Howell. Straightforward Java persistence through checkpointing. In R. Morrison, M. Jordan, and M. Atkinson, editors, *Advances in Persistent Object Systems*, pages 322–334. Morgan Kaufmann, 1999.

[20] Y. C. Hu, W. Yu, A. L. Cox, D. S. Wallach, and W. Zwaenepoel. Runtime support for distributed sharing in typed languages. In *Proceedings of LCR2000: the Fifth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Rochester, New York, May 2000.

[21] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, California, June 1992.

[22] M. Lal and R. Pandey. CPU resource control for mobile programs. In *International Symposium on Agent Systems and Applications*, Palm Springs, California, Oct. 1999.

[23] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, , and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 318–329, Montreal, Canada, June 1996.

[24] D. Malkhi, M. Reiter, and A. Rubin. Secure execution of Java applets using a remote playground. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 40–51, Oakland, California, May 1998.

[25] A. Marquez, J. N. Zigman, and S. M. Blackburn. A fast portable orthogonally persistent Java. *Software: Practice and Experience Special Issue: Persistent Object Systems*, 30(4):449–479, 2000.

[26] J. E. B. Moss. Design of the Mneme persistent object store. *ACM Transactions on Information Systems*, 8(2):103–139, 1990.

[27] K. Nilsen, S. Mitra, S. Sankaranarayanan, and V. Thanuvan. Asynchronous Java exception handling in a real-time context. In *IEEE Workshop on Programming Languages for Real-Time Industrial Applications*, Madrid, Spain, Dec. 1998.

[28] R. Pandey and B. Hashii. Providing fine-grained access control for Java programs. In R. Guerraoui, editor, *13th Conference on Object-Oriented Programming (ECOOP'99)*, number 1628 in Lecture Notes in Computer Science, Lisbon, Portugal, June 1999. Springer-Verlag.

[29] T. Printezis, M. P. Atkinson, L. Daynès, S. Spence, and P. Bailey. The design of a new persistent object store for PJama. In *Proceedings of the Second International Workshop on Persistence and Java (PJW2)*, Half Moon Bay, California, Aug. 1997.

[30] J. Protic, M. Tomaevic, and V. Milutinovic. *Distributed Shared Memory: Concepts and Systems*. IEEE Press, Aug. 1997. ISBN 0-8186-7737-6.

[31] A. Rudys and D. S. Wallach. Termination in language-based systems. *ACM Transactions on Information and System Security*, 5(2), May 2002.

[32] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 213–227, Seattle, Washington, Oct. 1996.

[33] A. Silbershatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. The McGraw-Hill Companies, Inc., 4th edition, 2002.

[34] E. G. Sirer, R. Grimm, A. J. Gregory, and B. N. Bershad. Design and implementation of a distributed virtual machine for networked computers. In *Proceedings of the Seventeenth ACM Symposium on Operating System Principles*, pages 202–216, Kiawah Island Resort, South Carolina, Dec. 1999. ACM.

[35] L. van Doorn. A secure Java virtual machine. In *Ninth USENIX Security Symposium Proceedings*, Denver, Colorado, Aug. 2000.

[36] D. S. Wallach, E. W. Felten, and A. W. Appel. The security architecture formerly known as stack inspection: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4):341–378, Oct. 2000.