

Transactions for Java

Alex Garthwaite and Scott Nettles

Computer and Information Science Department
University of Pennsylvania

September 9, 1996

Abstract

We present a design and implementation of transactions and general-purpose persistence for Java. These additions allow Java programmers to manipulate any Java object using transactions and provide resilience from machine failure for these objects. This extends the range of Java applicability into domains where reliability is of paramount concern; for example, network-based banking.

Our design and implementation is a significant addition to Java. It extends Java's current emphasis on safety and reliability to the safe and consistent management of permanent state.

Our additions take the form of syntactic extensions for transactions and runtime system support for durability and atomicity. Support for general-purpose persistence, the ability to arbitrary kinds of objects persistent, is a key aspect of the design. We provide orthogonal persistence, in which any object can be made persistent, without regard to type. We also provide persistence-by-reachability, in which an object becomes persistent if it is reachable from a special persistent root.

Our implementation is based on two key constructs: an extension to the Java heap to make parts of it persistent, and logging changes to this heap to support efficient commit and the ability to roll-back changes to support abort. Extensive details of the implementation are provided.

We have tested our system on a debt-credit benchmark, based on TPC-B. The result that our system can achieve a rate of 41 TPS, just under half that possible given the disks in use. We consider this a very acceptable results for an untuned system. Furthermore, our results suggest several avenues for improvement and it seem feasible to achieve transaction rates that are very close to the limits of our disks.

1 Introduction

This paper describes the design and implementation of transactions and general-purpose persistence for Java. This is a significant advance because it allows Java applications to reliably and consistently maintain permanent state, even in the face of machine failures. This support will allow Java to be used directly in applications for which reliability is of paramount concern; for example, in applications supporting financial transactions.

Java is a new but already popular programming language. Loosely speaking it is a "safe" C++. Its safety features include strong static type checking, the use of implicit storage management (garbage collection) to manage deallocation of dynamically allocated storage, and the absence of machine pointers at the language level. These features combine to make Java free of the kind of pointer errors that plague C++ programs; Java applications can never generate illegal pointers and thus never "dump core" because of illegal accesses. These safety properties are central to one of the main design goals of Java: the ability to safely transmit Java code across the Internet.

We have extended Java to support a new set of safety features; our goal is to support applications that need high reliability access to permanent data. Such applications range from the maintenance of personal calendars and contact databases to Web-based commerce. Our extensions take the form of support for transactions and general-purpose persistence.

In most systems, users can manipulate permanent data in very limited forms, typically files and various kinds of database records. General-purpose persistence allows any user defined data-type to be made permanent and thus resilient to machine failures. This feature is crucial to supporting transactions that manipulate arbitrary data because it allows the effects of transactions to be made permanent. Similarly, transactions are crucial when manipulating persistent data; if all updates changed the permanent state of a system then it would be very easy for crashes to allow inconsistent states to be observed; for example, it would be possible for a bank transfer to be interrupted in a state where money had been withdrawn from one bank account, but not yet deposited in another. It is exactly these sorts of inconsistencies that trans-

actions avoid.

Transactions and general-purpose persistence are well understood abstractions, it would be pointless to extend them here. Instead, we have focused on providing an implementation of these abstractions that is well integrated into Java. The key aspects of transactions that we focus on here are support for atomicity and durability.

Durability is provided by supporting general-purpose persistence and the atomic update of persistent objects on stable storage. Central to this support was making Java's heap persistent, with a representation on stable storage as well as in main memory. Also key are modifications to the Java Virtual Machine (VM) to log changes to the main memory heap and to then transfer these changes to the stable version atomically. Recoverable Virtual Memory [8] is used to provide low-level support for the persistent heap and its atomic update. The persistent heap implementation is closely integrated into the existing Java heap and uses Java's garbage collector to collect persistent storage.

We have performed basic benchmarks on our prototype using a debit-credit benchmark modeled closely on the industry standard TPC-B benchmark. The results show that we can achieve transaction rates of 41 TPS, about half the maximum possible with our disks. We believe this result can be improved to quite close to that of our disk. We have also compared the performance of the Java compiler on the original Java system and our modified version. These results show a small increase in overheads for our system in the compilation of some class libraries and minor improvements in others.

We first present the transaction and persistence model visible to the Java programmer. The bulk of the paper then focuses on the design and implementation, especially on how durability and atomicity is supported. Next, we discuss our performance evaluation, followed by related work. Finally, we discuss future work and conclude.

2 Programming Model

Transactions represent a significant extension to the Java language. One central question is what interface is presented to the user, and what semantics this interface supports. We take the conventional view [4] that transactions must provide the ACID features: atomicity, consistency, isolation, and durability. In our initial design, we have kept the transaction model simple and standard; this is because our primary concern is in supporting durability by allowing any Java object to be made persistent.

We will first discuss the basic transaction model and interface, followed by a somewhat more detailed discussion of persistence.

2.1 Transaction Model

We provide transactions as a syntactic extension to Java. This allows them to work with other similar control struc-

```
/* Assume my_obj and other_obj
   are persistent and each has
   an integer member named i.
 */

transaction {
    RW_Lock.acquire_write_lock(my_obj);

    my_obj.i = 1;
    RW_Lock.acquire_read_lock(other_obj);

    if ( other_obj.i == 1 ) {
        commit;
    } else {
        rollback;
    }
} /* End of transaction */
```

Figure 1: A simple transaction in Java

tures found in Java and provides a well defined scope for the rolling back of changes to variables in the thread stack. This means that statements such as `break`, `continue`, and `return` that affect the flow of control of the program may appear within a transaction even if their evaluation causes control to leave the transaction. This also allows transactions to work in concert with Java's exception handling mechanism. This syntax is fully integrated into the Java compiler, verifier, interpreter, and other tools.

As an example, Figure 1 shows a transaction that modifies a persistent integer and aborts or commits based on the value another persistent integer:

`Transaction` introduces a new scope and if the user explicitly aborts or commits, control is transferred to the end of that scope. If control simply reaches the end of the scope, or if an unhandled exception is raised, then a default action (usually `rollback`) is taken. Currently we do not allow nested transactions, but supporting them is a likely extension.

If the transaction aborts, which is syntactically denoted by `rollback`, the values of any objects modified by the transaction are rolled back to the values they had before the transaction began. This restoration includes the undoing of changes to values living in the thread stacks and static class members as well as those living in the heap. This provides atomicity in the face of failure. If the transaction commits then not only do the modified objects retain their new values, but those new values are also transferred to stable storage. Even if the machine crashes the new values will be retained, thus providing durability.

Our model allows for multiple top-level transactions, although their implementation is on-going. For concurrency control, the example shows that we use explicit locking based on reader-writer locking of whole objects. These locks are held until a transaction commits or aborts. As is usual, if a transaction attempts to acquire a read lock on an object that is held in write mode by another transaction, or if a transac-

tion attempts to acquire a write lock on an object that is held in either read or write mode by another transaction, then the transaction trying to acquire the lock blocks until the lock becomes free. These rules mean that transactions are isolated from each other. Furthermore, this locking strategy guarantees that transactions are serializable. We leave deadlock avoidance to the user, which is suitable for our basic prototype. It is likely that a production system would implement a deadlock detection system, but to do so here would only complicate our implementation without contributing to our research goals.

2.2 Persistence

If all objects modified by a transaction are to be durable, then it must be possible to make any object persistent. We call such support general-purpose persistence, in contrast to the special purpose persistence provided by file-systems or relational databases. Providing this feature is the focus of much of our current work.

In any system providing persistence, the question of which objects are persistent arises. One approach is to tie persistence to data-type, but this leads to a profusion of types and, in systems with static typing, makes it hard to use the same functions for both persistent and volatile objects. Instead, we provide orthogonal persistence [2] in which any object can be made persistent, without regard to type.

It is common for orthogonal persistence to be coupled to another feature, persistence-by-reachability, in which any object reachable by following pointers from a special persistent root is also persistent. The advantage of this feature is that it is impossible for a programmer to accidentally forget to make some part of a data structure persist, only to find out about this error at recovery time, when it may be much too late. Since garbage collection is also based on reachability, persistence-by-reachability is especially natural in garbage collected systems such as Java.

One final question arises concerning persistence. If the system crashes, how does the user locate the persistent objects after recovery from the crash? In our system we adopt a simple strategy. The persistent root is actually a symbol table. An object can be added to this table along with a name. Then the object becomes persistent (although it is not forced to stable storage until the next commit) as do any objects reachable from it. Upon recovery, the programmer can look the object up in the symbol table. The last committed value is returned. To support this functionality, the Persistence class has been added to the Java language package and contains the static methods listed in Figure 2.¹ `Mark` and `unmark` allow object to be added and removed from the root, while `lookup` allows objects to be looked up by name. `Restarted` is useful when code needs to tell if the system has been restarted and is often used in the initialization code

¹Note that the keyword `native` indicates that a method is written in C and not in Java.

```
// Add obj to the root indexed by key
static boolean mark(String key, Object obj);

// Remove object indexed by key from root
static void unmark(String key);

// Lookup the object indexed by key
static Object lookup(String key);

// True if the system has been restarted
static native boolean restarted();
```

Figure 2: Static methods provided by Persistence class

for static class members. For simplicity some other functions have been omitted, but the ones here provide the basic functionality.

3 Design and Implementation

In this section, we discuss the design and implementation of our system. The two central aspects of this section are our support for durability in the form of general-purpose persistence and our introduction of transactions into Java. Our implementation of multiple top-level transactions has not yet begun and we do not discuss its design here, nor do we discuss the implementation of reader/writer locks. However, the rest of our system is compatible with these features, and when appropriate we discuss how this compatibility is achieved.

In the Java interpreter, data is distributed among thread stacks, class structures, and a heap containing Java objects. In our implementation, both persistent and transitory data can be found in the class structures and the heap. The distinction between what is transitory and what is persistent is maintained by checking what data is reachable from a persistent set of roots. To keep a view of the persistent data in the heap, we maintain, in addition, a representation of this persistent data both in volatile memory and on stable storage (in our case, disk). If the system crashes, the stable representation of the heap is used to restore the representation in volatile memory. During a transaction, the user modifies data in volatile memory. In order to support commit for persistent data, we must be able to transfer these changes to the stable heap. These same changes must be rolled back in the event that a transaction aborts or the Java virtual machine is restarted on a persistent heap image containing uncommitted changes. Finally, storage in Java is managed automatically, and so we must provide for garbage collection of the persistent heap.

Transactions in our implementation track mutations to all data items whether they live on a thread stack, in some class structure, or in the heap. Rolling back the changes for an aborted transaction causes all of these changes to be undone. Together with the presence of similar syntactic struc-

tures in Java (e.g., synchronize and try statements), this is the primary reason for introducing the transaction statement as syntax: it defines a fixed scope within which changes are tracked. Transactions, as currently implemented, cannot be nested but we plan to add nested transactions in the near future.

3.1 Background

Our implementation is built upon Sun Microsystem's Java Developer's Source Release 1.0. Because of this starting point, it is important that we discuss some of the components that we used or modified in our implementation. They include the Java VM, the Java heap, and Recoverable Virtual Memory (RVM), which we use to manage the disk-based version of our heap.

3.1.1 The Java Virtual Machine (VM)

The Java Virtual Machine supports a pre-emptive concurrent threads model implemented using a user-level threads package. Currently kernel-level threads are not used. Each thread executes with its own stack and one thread cannot refer to any part of another thread's stack. In addition to the stacks, there are also structures for each currently loaded class including information about methods, fields, bytecode, class and superclass information, and constant pools. Finally, with the exception of local variables of base type (e.g., char, int, long, pointers to object handles), all other data objects live in the Java heap.

3.1.2 The Java heap and Garbage Collector

The Java heap provided with Sun's source release is organized into two areas: a handle space and an object space. Objects allocated in the heap are always accessed via a handle and the virtual machine currently does not allow pointers into the interior of objects. In addition to providing a pointer to the object's data, each handle also stores a pointer to the methods that that object's class supports. The methods are stored as part of the class structure which in the original release is not stored in the heap. Handles are used so that once allocated in the object space the object need not remain fixed and so that given an object the bytecode interpreter can identify the object's class.

The object space is divided into variable-length pieces with each piece containing a header describing the size of the chunk and whether it is currently free and the data for a single object. The headers for each chunk turn out to be important since no free list for the object space is kept. Instead, when a new object needs to be allocated, the object space is traversed by starting at a valid header, checking its state and size and using its size information to find the next header if that is necessary. This means that the headers must be kept consistent in order for allocation and garbage collection to work correctly. As we will see, it also presents problems

when a transaction is rolled back since other threads may have allocated objects in and around the ones allocated during that transaction. Finally, while each handle points to a single object, each object may be pointed at by any number of handles.

Garbage collection in Java is done using a non-concurrent mark-and-sweep technique. It may be invoked either synchronously when an allocation fails due to lack of a suitably sized free chunk or asynchronously when the garbage collection thread is scheduled during an idle period. In practice, however, all garbage collection occurs synchronously since the asynchronous GC thread will be pre-empted by any other event and almost never has time to complete a full collection. In the future, we hope to rectify this limitation by implementing a concurrent mark-and-sweep collector. Java's garbage collection is currently fairly expensive because it forces all other threads to suspend for the entire duration of the collection.

A garbage collection proceeds in three distinct phases:

- Mark all objects and handles referred to in the thread stacks, class structures, and other objects in the heap.
- Sweep through the object space marking unreferenced objects as free and coalescing contiguous free chunks.
- If necessary, compact the object space and update the appropriate handles.

If at any point during the sweep phase a contiguous block is found that is large enough to satisfy the request then the garbage collection ceases and the object is allocated there. Compaction of object space is possible because all objects are referenced through their handles. Unlike object space, handle space does not suffer from fragmentation because all handles are a fixed size. Because of the use of the object headers to connect object space in a singly-linked list, compaction proceeds by scanning from the beginning of object space both for free blocks and for possible blocks that can be moved. Also, since multiple handles may point to the same object, these handles are formed into a linked list at the start of the compaction phase and restored with the object's new address when the compaction process finishes.

3.1.3 RVM

Our design assumes the ability to make multiple updates to the stable version of the persistent heap atomically. To achieve this our implementation uses the Recoverable Virtual Memory system described by Satyanarayanan, et al. [8], although using RVM is not fundamental to our design. RVM provides simple non-nested transactions on byte arrays and uses a disk-based log for efficiency. RVM allows us to establish a one-to-one correspondence between parts of a file and ranges of virtual memory. RVM defines a simple interface that allows changes to the volatile heap to be transferred to the stable heap atomically. As part of its basic design, RVM

assumes that there will be sufficient main memory for all the persistent data; if this is not true, the system will page.

While using RVM has greatly simplified the construction of our implementation, there are two limitations that we have had to accept. The first is that RVM was designed for a very different kind of application—providing recovery services to a file-system—than the one we have built. The result is that it has not been optimized for use with Java and the overhead of maintaining some of its data structures can be quite high. An area of future work is to look into precisely what these overheads and whether we can reduce them. The second is that RVM does not provide for write-ahead logging.

3.2 Structure of the Persistent Heap

When considering how to add persistence to Java, we have chosen to implement a single heap containing both persistent and transitory data where persistence is a matter of being reachable from a set of persistent roots. Before choosing this approach, we considered and rejected several other possible alternative models: a two-heap model with one holding persistent data, and a single-heap model in which we tag the persistent objects and do not support reachability. We rejected the first option because too much of the Sun implementation depends on the fact that the handle and object spaces are each contiguous and because it would have required keeping track of cross-heap pointers. The second option is fairly simple to architect and does support orthogonal persistence in the sense that the property of being persistent remains independent of the type system. However, it complicates the task of the Java applet programmer since that person would need to be sure to make everything that needed to be persistent be persistent. In this section, we describe briefly what our model entails.

In order to support persistence, we have changed the VM and heap in several significant ways. First, we have modified the startup process so that the appropriate RVM structures are initialized. In the case of a initial persistent heap, this step is straightforward. When starting with an existing heap, however, we have to map in the heap from disk. In addition, we restore a number of data structures that the VM uses including a pointer to the list of persistent roots, the list of change logs for uncommitted transactions, the size and address of the heap, and where the list of class structures lives in the heap. We then proceed to map the heap in at the same address it was at before, roll back the effects of the outstanding transactions, mark the class structures as being uninitialized, and then proceed as usual.

Second, we have placed the class structures into the heap itself. There are two reasons for this change. The first reason is that we want to avoid the problem of classes changing in between invocations using the same persistent heap. If we do not keep class structures across invocations, we would need some mechanism to handle such a possibility. The second reason is that it simplifies the problem of how to reconnect objects and their handles to the appropriate parts of

the class structures when starting with an existing heap. By keeping the class structures in the heap and restoring these structures to the same addresses, this aliasing problem does not arise. However, because we put the class structures into the heap and these structures are C structures and not Java data structures and these structures are allocated using a malloc/free interface, we have complicated the way in which the heap can be maintained. We will return to this complication shortly.

Third, we have extended the set of bytecodes to include bytecodes to mark the start and end of transactions as well as to set the action to be taken when a transaction terminates. This change means that while our implementation can run bytecode generated by the Sun Java Development Kit, one must use our implementation to generate and run bytecode containing transactions.

Fourth, we have added support to the VM to track all changes to the heap and thread stacks created by Java. We have not extend this tracking to the case of the execution of native (C) routines. Here, we can offer suggestions for handling this case such as providing routines that would let the C routine register any changes—not a safe option—, use virtual memory protection to detect and log such changes during the execution of the native routine, or by “sandboxing” the native routine. One area for improvement in our implementation is that we currently track all changes in linked lists and perform no duplicate elimination. We hope to remedy this soon.

Finally, at any one time there are two persistent heaps on disk. One heap is used to store the Java heap as in its most recently committed state. The other heap is used when a garbage collection occurs to represent the heap after the collection completes. On recovery, the two heaps are distinguished by a field indicating which is the one most recently committed.

3.3 Logging of Modifications

For each outstanding transaction, a list of changes made during that transaction must be kept. This list is allocated and maintained in the heap and is marked persistent. Each change record is represented as a Java array of words and stores the address of the change as well as the old value. In the case of changes to heap objects, the address is to the handle of the object being changed together with an offset into that object. This approach enables us to let the compaction phase during garbage collection occur without worrying about having to fix up the addresses in the change lists afterwards. By representing the change lists as Java objects, we can release a list simply by setting the start of the list to null.

In addition to the user changes, the structure of the heap and the Java VM requires that we track other kinds of changes as well, even ones occurring outside of transactions. These changes include the loading of new class structures, the addition of new change lists, and the modification of handles and object headers in the heap. Without these changes

to VM meta-data, a consistent view of the heap would not be recoverable.

3.3.1 Java Virtual Machine

Modifications to data elements are captured in the bytecode interpreter just before the changes are made. In the case of the allocation of an object, we need to track the size of the object so that the whole object can be written out if the transaction commits and the object is persistent. In the case of a change to an existing value, we need to track where the change is and what the previous value is. Were we to implement write-ahead logging, we would also need to track the new value. However, we do not do so in this implementation because of limitations in RVM.

When a transaction starts, we create the necessary locking and change list structures for that transaction. Although we currently do not support either nested transactions or multiple concurrent transactions, the basic structures to support these features are in place in this implementation. Likewise, when a transaction block is exited and the end-of-transaction bytecode is executed, these lists are used in the manner appropriate for the intended action: commit or rollback.

In addition to tracking user modifications, modifications to meta-data in the virtual machine must also be recorded. This tracking lets us be able to restore the virtual machine and heap to a consistent state when starting with an existing persistent heap. Here are some of the changes to meta-data that we track. First, during the bootstrapping process where the VM starts up and enters the threads system, it does a number of allocations outside of the normal allocation mechanisms and these changes must be accounted for. Second, we track all changes to headers and handles, even ones outside of transactions, since allocations made during committed transactions will be made relative to these changes. If we do not log them, we would not be able to restore the heap with all the pieces of object space correctly accounted for. Fourth, we must commit all loaded classes to the heap so that, again, on recovery we will have access to the same classes should one of the persistent objects either be an instantiation of one of them or should it cause an object of that type to be created.

3.3.2 Commit

When a transaction commits, a number of actions occur. First, the set of objects reachable from the persistent roots is obtained. This set is used together with the change list for the transaction to determine what changes need to be written to the persistent heap on disk. Second, the meta-data about the VM needs to be updated to reflect the current set of loaded classes, outstanding transaction change lists, and similar information. Third, in the event that a garbage collection has occurred since the last commit, we flip the two heap images on disk. Fourth, a commit record needs to be generated to

RVM to ensure the modifications are actually on disk. Finally, the change list for the committing transaction must be released. Since the Java VM meta-data, in particular, must be consistent, a mutex is taken on the heap for the duration of the commit process.

The meta-data for the Java VM must be written out to persistent storage along with the other changes. Since the header and handle changes are logged to RVM at the time they occur, no further action is required to register them. Likewise, the changes for classes that are loaded are also logged to RVM when they occur. Other than ensuring exclusive access to the heap to guarantee that the heap is in a steady and consistent state for the duration of the commit, no further action is required.

3.3.3 Rollback

When a transaction is rolled back, however, we need only roll back the changes in the change list, in order, free up the change list, and release any locks held during the transaction. No changes are logged to RVM and so no corrective actions are required with the persistent copy on disk.

Unlike user modifications, the changes to headers and handles cannot be completely undone. Instead, we use the change list for the transaction to determine which ones have been changed and mark these as free. Since these are only changed when an allocation occurs, this action is safe. In addition, it means that the headers in object space remain consistent even if other threads have interspersed allocations with the ones made during the aborted transaction.

3.4 Persistent Heap Garbage Collection

With the exception of an additional phase, garbage collection in our implementation is largely unchanged. It remains a non-concurrent mark-and-sweep collector with same basic set of phases. During the first phase, we now also sweep the C data structures that have been moved into the heap. We can accomplish this task in a precise manner because we know that all such structures are part of class structures and we know how these are built and maintained. In the future, we may change this approach to a conservative one if we find it useful to put other C structures into the heap. We also sweep the lists of change records for uncommitted transactions to make sure that nothing they refer to is left unmarked and considered garbage. The other phase that changes is the compaction phase. Here, we must make sure to pin the C data structures in the heap so that they are not moved. This pinning is done because these structures refer to each other directly and not through the use of handles. If we did not prevent them from being moved, we would have to introduce a mechanism to repair any pointers they contain.

When the collection finishes, we write out the newly collected heap to the secondary heap image on disk and record that at the next commit this heap should become the primary

committed heap image. This flip is implemented by incrementing a counter in the meta-data header of the secondary heap image and setting a flag so that, at the next commit, the flip occurs. Currently, writing the heap is done synchronously. One improvement we plan is to allow this process to proceed asynchronously and have the commit process wait for it to finish if needed.

4 Performance Evaluation

In this section we discuss our preliminary evaluation of the performance of our system. First, we discuss the benchmarks used and the machine environment. Then we discuss the results of a TPC-B-like debit-credit benchmark, which allows us to investigate the performance of our basic transaction mechanisms. Finally, we discuss the results of comparing the performance of the Java compiler using our system and the unmodified version. This demonstrates that our changes have no ill-effects on basic Java performance in the absence of transactions.

4.1 Experimental Setup

Two issues involving our experimental setup are important: the benchmarks we used, and the basic machine environment.

4.1.1 Benchmarks

To study the transaction processing performance of our system, we wrote a simple Java program based on the TPC-B benchmark [10]; because we do not follow the TPC-B standard, we call this benchmark Debit-Credit. Debit-Credit uses our transaction extensions to model a simple banking system; see the TPC-B standard [10] for details. After performing a small amount of computation the benchmark commits its results to stable storage; thus it is almost exclusively a test of the performance of our low-level support for durability, and of course the performance of our disks.

The only way our benchmark differs from TPC-B is that the size of the database is scaled to fit in main memory. This is in keeping with the basic design of RVM. Because of this, the benchmark does not test how effectively we can retrieve data from disk. The benchmark is single-threaded, so there is little variance in the time for each commit, since each one does the same amount of work and then commits.

To test whether our changes had any significant impact on Java's basic performance, we used a version of the Java compiler that included our changes, and one that was unmodified. The actual test involved compiling the Java class libraries, totaling about 90 thousand lines of code.

Quantity measured	TPS	Elapsed (msec)	User CPU (msec)	Sys CPU (msec)
Complete	40.8	24.5	10.21	0.63
SysMalloc	41.9	23.9	9.84	0.62
Writes Only		13.2	0.12	0.77
No Writes	94.5	10.6	10.26	0.12
No RVM	307	3.26	3.01	0.11
No Durability	1571	0.642	0.43	0.106

Table 1: Debit-Credit

4.1.2 Machine Environment

To test our system, we used a SPARCsystem 20, running SunOS 5.4. The machine uses a superSPARC CPU running at 50 MHz and had 32 megabytes of memory. To avoid interference from the file-system and achieve maximum performance, we used a dedicated raw disk for RVM's log. This disk rotates at 5400 RPM, and thus has a basic rotational delay of about 11 milliseconds. We wrote a simple program that just writes synchronously to the disk. This program was able to write the disk at a rate equivalent to 89 TPS, thus bounding the maximum performance our system might achieve.

4.2 Debit-Credit

Table 1 shows the results of running Debit-Credit, both in terms of the number of transactions per second, and in terms of overhead in milliseconds per transaction, both for elapsed time and user and system CPU time. The row labeled *Complete* is for the complete system. The row labeled *SysMalloc* is for the system without forcing all mallocs to occur in the garbage collected heap. The row labeled *Writes Only* gives the results of timing just the synchronous writes done by RVM. The row labeled *No Writes* measures the cost if the data is discarded at the point it would normally be written. The row labeled *No RVM* has the results without any calls to RVM. Finally, the row labeled *No Durability* shows the results when all the overhead for persistence is eliminated; these results do include the basic Java overheads as well as the cost of the features that support roll-back. All of these times are the result of averaging over the results of five runs.

The basic result is our full system achieves just under 41 TPS, which is 46% of the maximum achieved when just writing synchronously to the disk. The difference between the elapsed time and the CPU time gives us the time spent in I/O, 13.7 msec. We consider this a more than acceptable result, especially considering that we have not tuned our system at all. The other results provide some important clues as to how to tune our system.

First, the performance when mallocing in the garbage collected heap and when using SysMalloc are essentially the same. This implies that our changes to these aspects of the system have had no ill effects on the performance. Next, notice that the *Write Only* case takes just a bit longer than

a disk rotation, 11 milliseconds, and almost all of this time is spent doing I/O. This suggests that on average we miss one rotation during the write. Also notice that time spent when there are no writes is just under the time for a rotation and not surprisingly has essentially no I/O. From this and the fact that the complete time is about two rotational delays, we conclude that in general we are able to commit a transaction every two rotations of the disk. If we are able to bring the CPU overhead of our system (the no write case) down such that it plus the CPU time involved in actually doing the write is less than a rotational delay, then we should see almost a factor of two improvement, since we will avoid missing a disk revolution. We have observed such a speedups before using similar benchmarks [9].

For clues about how to reduce our overheads, we first note that the overheads without durability are minimal, thus speeding up Java or making write logging faster will have little effect. The key is to speedup the persistence system in the runtime. The difference between the *No Durability* and the *No RVM* is the overhead of the code we wrote to support persistence, about 2.6 milliseconds. The overhead from RVM is the difference between the *No Writes* and *No RVM* cases, about 7.3 milliseconds.

These results suggest that if we want to speed up our system, we should either attempt to reduce the cost of RVM, or use RVM less. Both of these are possible. RVM was coded with the assumption that programmers would use it directly, and thus it does a great deal of error checking. We only need this error checking for debugging. Also, currently we must log changes to the headers in Java's heap. An alternative strategy for recreating these headers upon recovery could easily let us avoid these overheads.

4.3 Java Compiler

When we ran the Java compiler on the Java class libraries using our system and the original system, we obtained the results shown in Table 2, which represent the averages of 5 runs on an unloaded machine. This suggests that our system adds a small overhead, essentially all of which is extra CPU time. We are currently attempting to find the source of this overhead.

5 Related Work

Transactions were first developed by the database community and have an extensive literature. We do not attempt to review this literature here, but rather refer the reader to Gray [4].

More relevant to the current work is the literature concerning persistent programming languages, in particular languages supporting orthogonal persistence and persistence-by-reachability. The pioneering work is that of PS-Algol [2] and its follow-up language, Napier88 [6]. These languages support much the same persistence models provided here, as

Library	Compiler	Elapsed Time (secs)	CPU Time (secs)	System Time (secs)
java AWT	Original	72.85	59.48	4.92
	Modified	79.77	66.86	4.86
applet	Original	4.96	3.30	0.97
	Modified	5.04	3.53	0.986
java net	Original	10.30	7.77	1.33
	Modified	10.83	8.50	1.35
javadoc	Original	33.11	28.86	3.01
	Modified	36.48	32.25	3.06
debug	Original	179.60	169.74	5.22
	Modified	198.88	188.37	5.24
audio	Original	9.11	6.88	1.22
	Modified	10.20	7.42	1.35
mmedia	Original	56.31	44.17	7.80
	Modified	61.91	49.60	7.76
sun net	Original	41.34	30.13	5.19
	Modified	44.86	33.17	5.27
sun TAWT	Original	45.56	35.84	6.16
	Modified	49.50	39.89	6.33
sun applet	Original	28.38	22.33	3.88
	Modified	31.14	24.71	4.20
Total	Original	481.52	408.50	39.71
	Modified	528.61	454.31	40.42

Table 2: Compilation of Java libraries

do many other persistent languages. We make no claim of novelty for our interface, our contribution comes in the implementation of a well tested interface in Java.

Prior to this work, one of the authors participated in the Venari project [5], of which one goal was to provide transactions for Standard ML (SML). A discussion of the low-level aspects that system (called Sidney) can be found in Nettles [9]. Not surprisingly, our current work draws strongly on this prior work. There are some key differences that arise from the different languages involved, but they are not particularly relevant here. One advantage of SML worth mentioning is that its support for higher-order functions allowed transaction to be added without adding new syntax. This resulted in a much less complex implementation, without any loss of power. In fact, transactions in Venari are much more flexible than those provided here.

It is useful to briefly compare the two implementations. Both use RVM for their low-level transaction services, and as such inherit some of RVMs restrictions, in particular the need to map all persistent data into memory. They also both use write-logging by the language system (in-lined by the compiler for SML) to provide much of the information needed by commit and abort. However, major differences are found in the structure of the heaps and in the nature of their garbage collectors. Unlike the current work, Sidney uses an explicitly separate persistent heap, and upon commit must move objects that have become reachable from the persistent root from the transitory heaps into the persistent heap. This

is not as difficult as it sounds because SML/NJ and Sidney use copying garbage collection and thus moving objects is quite natural. The mark-and-sweep nature of Java heap has dictated much different design choices, and at a detailed level there is little similarity with Sidney. Finally, Sidney provides for concurrent collection of its persistent heap, which allows collections to be done in a manner that is non-disruptive to the client. Sidney is the first implementation to provide this functionality for a transactional heap [7].

One previous concurrent mark-and-sweep collector for persistent data has been implemented by Almes for the Hydra system [1]. Hydra did not support transactions or orthogonal persistence, and it is not clear exactly what guarantees it offered to the client about data consistency. The lack of support for transactions means that Almes collector could essentially be a standard concurrent mark-and-sweep collector. (In fact, it is likely that it was the first such collector.) However this lack means that at the least Almes' techniques will have to be significantly extended to support concurrent collection in the current case.

We are aware of one other effort to provide general-purpose persistence for Java [3]. Since this work will be discussed in detail at this workshop, we omit a detailed discussion here.

6 Status and Future Work

Our implementation is on-going. The next goal will be to support multiple transactions, the key hooks are already present for these, most of the remaining work lies in providing reader/writer locks and in managing multiple write-logs. Nested transactions are simple to implement using write logs, as shown by the Venari work [5]. These are all of the semantic features we expect to support.

We also expect to perform more extensive performance measurements. This should help us better characterize the system and help us improve its performance. In particular, we expect better understand which of the possibilities mentioned in the performance section will let us improve our basic transaction performance and then to study the effect of making those improvements.

The most difficult piece of future work is in providing a concurrent garbage collector for Java and in making that collector work properly for persistent data in the presence of transactions. Concurrent mark-and-sweep technology is well understood, but we are not aware of an implementation that supports transactions. Our design for this work is relatively complete, but we have not begun an implementation.

7 Conclusions

We have described a transaction system for Java. It supports general-purpose persistence, and thus allows any Java object to be manipulated transactionally. Our implementation

is based on making parts of Java's heap persistent and on logging changes to the heap to support both commit and roll-back. The results of our preliminary performance evaluation are quite promising. Without tuning, our system can reach about half of the maximum possible performance given our disk. Possibilities for improving the performance are substantial.

References

- [1] G. T. Almes. Garbage Collection in a Object-Oriented System. Technical Report CMU-CS-80-128, Carnegie Mellon University, June 1980.
- [2] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An Approach to Persistent Programming. *Computer Journal*, 26(4):360–365, December 1983.
- [3] Malcolm. P Atkinson. Personal communication.
- [4] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan-Kaufmann, 1993.
- [5] N. Haines, D. Kindred, J. G. Morrisett, S. M. Nettles, and J. M. Wing. Composing First-Class Transactions. *ACM Trans. on Prog. Lang. and Systems, Short Communications*, 16(6):1719–1736, 1994.
- [6] R. Morrison, A. L. Brown, R. Carrick, R. Conner, and A. Dearle. On the Integration of Object-Oriented and Process-Oriented Computation in Persistent Environments. In *Advances in Object-Oriented Database Systems*, pages 334–339, 1988.
- [7] J. O'Toole, S. Nettles, and D. Gifford. Concurrent Compacting Garbage Collection of a Persistent Heap. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. ACM, December 1993.
- [8] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight Recoverable Virtual Memory. *ACM Transactions on Computer Systems*, 12(1):33–57, February 1994. Corrigendum: *ACM Transactions on Computer Systems*, 12(2):165–172, May 1994. Also available in *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, December 1993.
- [9] Scott Nettles. Safe and Efficient Persistent Heaps. Technical Report CMU-CS-TR-95-225, Carnegie Mellon School of Computer Science, December 1995.
- [10] Transactions Processing Council. TPC-B. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Processing Systems*, pages 79–114. Morgan-Kaufmann, 1991.