Noninvasive concurrency with Java STM

Guy Korland[1] and Nir Shavit[1] and Pascal Felber[2]

[1] Computer Science Department
Tel-Aviv University, Israel,
guykorla@post.tau.ac.il, shanir@post.tau.ac.il

[2] Computer Science Department
University of Neuchâtel, Switzerland,
pascal.felber@unine.ch

Abstract. In this paper we present a complete, compiler independent, Java STM framework called DEUCE, intended as a development platform for scalable concurrent applications and as a research tool for designing STM algorithms. DEUCE provides several benefits over existing Java STM frameworks: it avoids any changes or additions to the JVM, it does not require language extensions or intrusive APIs, and it does not impose any memory footprint or GC overhead. To support legacy libraries, DEUCE dynamically instruments classes at load time and uses an original "field-based" locking strategy to improve concurrency. DEUCE also provides a simple internal API allowing different STMs algorithms to be plugged in. We show empirical results that highlight the scalability of our framework running benchmarks with hundreds of concurrent threads. This paper shows, for the first time, that one can actually design a Java STM with reasonable performance, without compiler support.

1 Introduction

Multicore CPUs have become commonplace, with dual-cores powering almost any modern portable or desktop computer, and quad-cores being the norm for new servers. While multicore hardware has been advancing at an accelerated pace, software for multicore platforms seems to be at a crossroads.

Currently, two diverging programming methods are commonly used. The first exploits concurrency by synchronizing multiple threads based on locks. This approach is well known to be a two-edged sword: on the one hand, locks give the programmer a fine-grained way to control the applications critical sections, allowing an expert to provide great scalability; on the other hand, because of the risk of deadlocks, starvation, priority inversions, etc., they impose a great burden on non-expert programmers, often leading them to prefer the safer but non-scalable alternative of coarse-grained locking.

The second method is a shared-nothing model common in Web based architectures. The applications usually contain only the business logic, deployed in a container, while the state is saved in an external multi-versioned control system, such as database, message queue, or distributed cache. While this method removes the burden of handling concurrency in the application, it imposes a huge performance impact on data accesses and, for many types of applications, is difficult to apply.

The hope is that transactional memory (TM) [11] will simplify concurrent programming by combining the desirable features of both methods, providing *state-aware shared memory* with *simple concurrency control*.

In the past several years there has been a flurry of design and implementation work on the software transactional memory (STM) [21]. However, the state of the art today is less than appealing. With the notable exception of transactional C/C++ compilers [15], most STM initiatives have remained academic experiments, applicable only to "toy applications", and though we have learned much from the process of developing them, they have not reached a state that will allow them to be seriously field tested. There are several reasons for this. Among them are the problematic handling of many features of the target language, the large performance overheads, and the lack of support for legacy libraries. Moreover, many of the published results have been based on prototypes whose source code is unavailable or poorly maintained, making a reliable comparison between the various algorithms and designs very difficult.

In this paper, we introduce DEUCE, our novel open-source Java framework for transactional memory. DEUCE has several desired features not found in earlier Java STM frameworks. As we discuss in Section 2, there currently does not exist an *efficient* Java STM framework that delivers a full set of features and can be added to an existing application *without changes* to its compiler or libraries. It was not clear if one could build such an an efficient "compiler independent" Java STM.

DEUCE is intended to fill this gap. It is non-intrusive in the sense that no modifications to the Java virtual machine (JVM) or extensions to the language are necessary. It uses, by default, an original locking design that detects conflicts at the level of individual *fields* without a significant increase in the memory footprint (no extra data is added to any of the classes) and therefore there is no GC overhead. This locking scheme provides finer granularity and better parallelism than former object-based lock designs. DEUCE provides weak atomicity, i.e., does not guarantee that concurrent accesses to a shared memory location from both inside and outside a transaction are consistent. This is in order to avoid a performance penalty. Finally, it supports a pluggable STM back-end and an open and easily extendable architecture, allowing researchers to integrate and test their own STM algorithms within the DEUCE framework.

DEUCE has been heavily optimized for efficiency and, while there is still room for improvements, our performance evaluations on several high-end machines (up to 128 hardware threads on a 16-core Sun Niagara-based machine and a 96-core Azul machine) demonstrate that it scales well. Our benchmarks show that it outperforms the main competing JVM-independent Java STM, the DSTM2 framework [9], in many cases by two orders of magnitude, and in general scales well on many workloads. This paper shows for the first time that one can actually design a Java STM with reasonable performance without compiler support.

The DEUCE framework has been in development for more than two years, and we believe it has reached a level of maturity sufficient to allow it to be used by developers with no prior expertise in transactional memory. It is our hope that DEUCE will help democratize the use of STMs among developers, and that its open-source nature will encourage STM them to extend the infrastructure with novel features, as has been the case, for instance, with the Jikes RVM [1].

The rest of the paper is organized as follows. We first overview related work in Section 2. Section 3 describes DEUCE from the perspective of the developer of a concurrent application. In Section 4 we discuss the implementation of the framework, and then show how it can be extended, by the means of the STM backends, in Section 5. Section 6 presents a companion front-end to DEUCE that supports transactional Java language extensions. Finally, in Section 7, we evaluate the performance of DEUCE.

2 Related work

Several tools have been proposed to allow the introduction of STM into Java. They differ in their programming interface and in the way they are implemented. One of the first proposals is Harris and Fraser's CCR [8] that used a C-based STM implementation underneath the JVM and a programmatic API to use STM in Java.

DSTM [10] is another pioneering Java STM implementation. It used an explicit API to demarcate transactions and read/write shared data. Transactional objects had to additionally provide some pre-defined functionality for cloning their state.

More recently, the DSTM2 [9] framework was introduced, proposing a set of higher-level mechanisms to support transactions in Java. DSTM2 is a pure Java library that does not require any change to the JVM nor to the compiler. It uses a special annotation to mark an atomic interface. Only a class that implements an annotated interface can participate in a transaction. This class is created using a special factory and it can only support transactional access to primitive types or other annotated classes. Transactions must be started using a `Callable` object. This design creates an API that, while powerful and elegant, requires important refactoring of legacy code, and introduces many limitations, such as the lack of support for existing libraries.

The LSA-STM [19] is a Java STM that also relies primarily on annotations for TM programming. Transactional objects, to be accessed in the context of a transaction, are annotated as `@Transactional`, and methods are declared as `@Atomic`. Additional annotations can be used to indicate that some methods will not modify the state of the target object, (read-only) or to specify the behavior in case an exception is thrown within a transaction. Transactional objects are implemented in the LSA-STM using inheritance and must support state cloning. As such it does not fully support legacy code, and unlike DEUCE is not transparent or non-invasive. Among the limitations of the framework, one can mention that accesses to a field are only supported as part of methods from the owner class. Therefore, public and static fields cannot be accessed transactionally. Note that this limitation also applies to other Java STM implementations that use object-level conflict detection such as DSTM and DSTM2.

AtomJava [12] takes a different approach to Java STM design by providing a source-to-source translator based on Polyglot [16], an extensible compiler framework. AtomJava adds a new `atomic` keyword to mark atomic blocks, and performs some major extensions during code transformation, such as adding an extra instance field to each and every class. AtomJava relies on this field to maintain an object-based lock schema. This schema is shown to reduce lock access overhead, but imposes a memory overhead which impacts not only transactional objects but all the application objects. The source code translation approach simplifies the tuning and verification of STM instrumentation, but it makes it harder for the programmer to debug her original

code. Also, by translating source code, AtomJava imposes a major limitation on users in that it prevents them from using compiled libraries, and renders their source code with atomic blocks incompatible with regular Java compilers (unlike approaches such as DSTM2 and LSA-STM that are based on annotations).

Multiverse [14] is another STM implementation for Java that performs instrumentation driven by annotations. Similarly to DEUCE, Multiverse supports field-based conflict detection granularity, but fields can be monitored as part of a transaction only if their owner class was annotated as `@TransactionalObject`. This requires the developer to be aware of all the data structures that might be accessed as part of a transaction. Nevertheless, this approach provides Multiverse with the ability to maintain strong atomicity by preventing any access to transactional object members that is not part of a transaction's context. Multiverse provides several other features, such as a `retry`/`orelse` construct, different levels of optimistic and pessimistic behavior, and limited commuting operations.

Modifications to the JVM have been proposed [22] to replace Java monitors by transactions. Atomos [4] is a Java extension that replaces the `synchronized` keyword by `atomic`, and `wait`/`notify` operations by `retry` statements. Such conversions are quite complex, because the transformed code must maintain the same semantics as the original code, and has to support operations like irrevocable actions or signaling. In the end, the decision to replace a monitor depends on the tradeoff between STM overhead and the gain in parallelism.

There exist several other STM implementations for various programming languages. An exhaustive list is outside of the scope of this paper, but the interested readers can consult the online TM bibliography available from the Web page at `http://www.cs.wisc.edu/trans-memory/biblio/index.html`.

3 Concurrent Programming with DEUCE

One of the main goals in designing the DEUCE API was to keep it simple. A survey of the past designs (see previous section) reveals three main approaches: (i) adding a new reserved keyword to mark a block as atomic, e.g., `atomic` in AtomJava [12]; (ii) using explicit method calls to demarcate transactions and access shared objects, e.g., DSTM2 [9] and CCR [8]; or (iii) requiring atomic classes to implement a predefined interface or extend a base class, and to implement a set of methods [10]. The first approach requires modifying the compiler and/or the JVM, while the others are intrusive from the programmer's perspective as they require significant modifications to the code (even though some systems use semi-automatic weaving mechanisms such as aspect-oriented programming to ease the task of the programmer, e.g., [19]).

In contrast, DEUCE has been designed to avoid any addition to the language or changes to the JVM. In particular, no special code is necessary to indicate which objects are transactional, no method needs to be implemented to supporting transactional objects (e.g., cloning the state as in [10]). This allows DEUCE to seamlessly support transactional accesses to compiled libraries. The only piece of information that must be specified by the programmer is, obviously, which part of the code should execute atomically in the context of a transaction.

To that end, DEUCE relies on Java annotations. Introduced as a new feature in Java 5, annotations allow programmers to mark a method with metadata that can be

consulted at class loading time. DEUCE introduces new types of annotations to mark methods as atomic: their execution will take place in the context of a transaction.

This approach has several advantages, both technically and semantically. First technically, the smallest code block Java can annotate is a method, which simplifies the instrumentation process of DEUCE and provides a simple model for the programmer. Second, atomic annotations operate at the same level as *synchronized* methods, which execute in a mutually exclusion manner on a given object; therefore, atomic methods provide a familiar programming model.

```
1 public int sum(List list) {
2   int total = 0;
3   atomic {
4     for (Node n : list)
5       total += n.getValue();
6   }
7   return total;
8 }
```

Fig. 1. Atomic block example.

From a semantic point of view, implementing atomic blocks at the granularity of methods removes the need to deal with local variables as part of the transaction. In particular, since Java doesn't allow any access to stack variables outside the current method, the STM can avoid logging many memory accesses. For instance, in Figure 1, a finer-granularity atomic block would require costly logging of the `total` local variable (otherwise the method would yield incorrect results upon abort) whereas no logging would be necessary when considering atomic blocks at the granularity of individual methods. In cases when finer granularity is desirable, DEUCE can be used in combination with the TMJAVA front-end discussed in Section 6.

To illustrate the use and implementation of DEUCE, we will consider a well-known but non-trivial data structure: the skip list [17]. A skip list is a probabilistic structure based on multiple parallel, sorted linked lists, with efficient search time in $\mathcal{O}(\log n)$.

Figure 2 shows a partial implementation of skip list, with an inner class representing nodes and a method to search for a value through the list. The key observation in this code is that transactifying an application is as easy as adding `@Atomic` annotations to methods that should execute as transactions. No code needs to be changed within the method or elsewhere in the class. Interestingly, the linked list directly manipulates arrays and accesses public fields from outside their enclosing objects, which would not be possible with DSTM2 or LSA-STM.

One can also observe that the `@Atomic` annotation provides one configurable attribute, `retries`, to optionally limit the amount of retries the transaction attempts (at most 64 times in the example). A `TransactionException` is thrown in case this limit is reached. Alternatively one can envision providing timeout instead (Which we might add in future versions).

A DEUCE application is compiled with a regular Java compiler. Upon execution, one needs to specify a Java agent that allows DEUCE to intercept every class loaded and manipulate it before it is loaded by the JVM. The agent is simply specified on the command line as a parameter to the JVM, as follows:

```
1  public class SkipList {
2    private static class Node {
3      public final int value;
4      public final Node[] forward;
5      // ...
6      public Node(int level, int v) {
7        value = v;
8        forward = new Node[level + 1];
9      }
10     //...
11   }
12   private static int MAX_LEVEL = 32;
13   private int level;
14   private final Node head;
15   // Continued in next column...
```

```
16   @Atomic(retries=64)
17   public boolean contains(int v) {
18     Node node = head;
19     for (int i = level; i >= 0; i--) {
20       Node next = node.forward[i];
21       while (next.value < v) {
22         node = next;
23         next = node.forward[i];
24       }
25     }
26     node = node.forward[0];
27     return (node.value == v);
28   }
29   // ...
30 }
```

Fig. 2. @Atomic method example.

```
java -javaagent:deuceAgent.jar MainClass args...
```

As will be discussed in the next section, DEUCE instruments every class that may be used from within a transaction, not only classes that have `@Atomic` annotations. If it is known that a class will never be used in the context of a transaction, one can prevent it from being instrumented by providing exclusion lists to the DEUCE agent. This will speed up the application loading time yet should not affect execution speed.

4 DEUCE Implementation

This section describes the implementation of the DEUCE framework. We first give a high-level overview of its main components. Then, we explain the process of code instrumentation. Finally, we describe various optimizations that enhance the performance of transactional code.

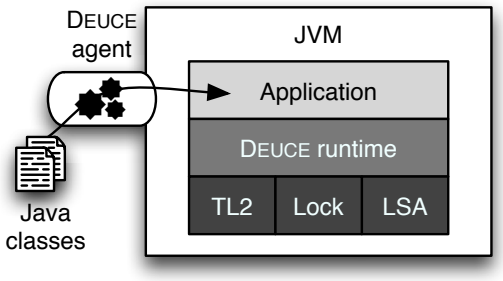

Fig. 3. Main components of the DEUCE architecture.

4.1 Overview

The DEUCE framework is conceptually made up of 3 layers, as shown in Figure 3:

1. The application layer, which consists of user classes written without any relationship to the STM implementation, except for annotations added to atomic methods.

2. The DEUCE runtime layer, which orchestrates the interactions between transactions executed by the application and the underlying STM implementation.
3. The layer of actual STM libraries that implement the DEUCE context API (see Section 5), including a single-global-lock implementation (denoted as "Lock" in the figure, and used as a performance "reference point" for all other libraries).

In addition, the DEUCE agent intercepts classes as they are loaded and instruments them before they start executing.

4.2 Instrumentation Framework

DEUCE's instrumentation engine is based on ASM [3], an all-purpose Java bytecode manipulation and analysis framework. It can be used to modify existing classes, or to dynamically generate classes, directly in binary form. The DEUCE Java agent uses ASM to dynamically instrument classes as they are loaded, before their first instance is created in the virtual machine. During instrumentation, new fields and methods are added, and the code of transactional methods is transformed. We now describe the different manipulations performed by the Java agent.

Fields. For each instance field in any loaded class, DEUCE adds a synthetic constant field (`final static public`) that represents the relative position of the field in the class. This value, together with the instance of the class, uniquely identifies a field, and is used by the STM implementation to log field accesses.

```
1 public class SkipList {
2   public static final long __CLASS_BASE__ = ...
3   public static final long MAX_LEVEL__address__ = ...
4   public static final long level__address__ = ...
5   // ...
6   private static int MAX_LEVEL = 32;
7   private int level;
8   // ...
9 }
```

Fig. 4. Fields address.

Static fields in Java are effectively fields of the enclosing class. To designate static fields, DEUCE defines for each class a constant that represents the base class, and can be used in combination with the field position instead of the class instance.

For instance, in Figure 4, the `level` field is represented by `level__ADDRESS__` while the `SkipList` base class is represented by `__CLASS_BASE__`.

Accessors. For each field of any loaded class, DEUCE adds synthetic `static` accessors used to trigger field's access events on the local context.

Figure 5 shows the synthetic accessors of class `SkipList`. The *getter* `level__Getter$` receives the current field value and a context, and triggers two events: `beforeReadAccess` and `onReadAccess`; the result of the latter is returned by the getter. The *setter* receives a context and yields a single event: `onWriteAccess`. The reason for having two events in the getter is technical: the "before" and "after" events allow the STM

```
1   public class SkipList {
2     private int level;
3     // ...
4
5     // Synthetic getter
6     public int level__Getter$(Context c) {
7       c.beforeReadAccess(this, level__ADDRESS__);
8       return c.onReadAccess(this, level, level__ADDRESS__);
9     }
10    // Synthetic setter
11    public void level__Setter$(int v, Context c) {
12      c.onWriteAccess(this, v, level__ADDRESS__);
13    }
14    // ...
15  }
```

Fig. 5. Fields accessors.

backend to verify that the value of the field, which is accessed between both events without using costly reflection mechanisms, is consistent. This is typically the case in time-based STM algorithms like TL2 and LSA that ship with DEUCE.

```
1   private static class Node {
2     public Node[] forward;
3     // ...
4
5     // Original method
6     public void setForward(int level, Node next) {
7       forward[level] = next;
8     }
9
10    // Synthetic duplicate method
11    public void setForward(int level, Node next, Context c) {
12      Node[] f = forward__Getter$(c) {
13      forward__Setter$(f, level, c)
14    }
15    // ...
16  }
```

Fig. 6. Duplicate method.

Duplicate methods. In order to avoid any performance penalty for non transactional code, and since DEUCE provides weak atomicity, DEUCE duplicates each method to provide two distinct versions. The first version is identical to the original method: it does not trigger an event upon memory access and, consequently, does not impose any transactional overhead. The second version is a synthetic method with an extra `Context` parameter. In this instrumented copy of the original method, all field accesses (except for `final` fields) are replaced by calls to the associated transactional accessors. Figure 4 shows the two versions of a method of class `Node`. The second synthetic overload replaces `forward[level] = next` by calls to the synthetic getter and setter of the `forward` array. The first call obtains the reference to the array object, while the second one changes the specified array element (note that getters and setters for array elements have an extra index parameter).

```
1  public class SkipList {
2    // ...
3
4    // Original method instrumented
5    public boolean contains(int v) {
6      Throwable throwable = null;
7      Context context =
8        ContextDelegator.getInstance();
9      boolean commit = true;
10     boolean result;
11
12     for (int i = 64; i > 0; --i) {
13       context.init();
14       try {
15         result = contains(v, context);
16       } catch(TransactionException ex) {
17         // Must rollback
18         commit = false;
19       } catch(Throwable ex) {
20         throwable = ex;
21       }
22       // Continued in next column...
```

```
23       // Try to commit
24       if (commit) {
25         if (context.commit()) {
26           if (throwable == null)
27             return result;
28           // Rethrow application exception
29           throw (IOException)throwable;
30         }
31       } else {
32         context.rollback();
33         commit = true;
34       }
35     } // Retry loop
36     throw new TransactionException();
37   }
38
39   // Synthetic duplicate method
40   public boolean contains(int v, Context c) {
41     Node node = head__Getter$(c);
42     // ...
43   }
44 }
```

Fig. 7. Atomic method.

Atomic methods. The duplication process described above has one exception: a method annotated as `@Atomic` does not need the first uninstrumented version. Instead, the original method is replaced by a transactional version that calls the instrumented version from within a transaction that executes in a loop. The process repeats as long as the transaction aborts and a bound on the number of allowed retries is not reached. Figure 7 shows the transactional version of method `contains`.

4.3 Summary

To summarize, DEUCE performs the following instrumentation operations on the Java bytecode at load-time.

- For every field, a constant is added to keep the relative location of the field in the class for fast access.
- For every field, 2 accessors are added (getter and setter).
- For every class, a constant is added to keep the reference to the class definition and allow fast access to static fields.
- Every (non-`@Atomic`) method is duplicated to provide an atomic and a non-atomic version.
- For every `@Atomic` method, an atomic version is created and the original version is replaced with a retry loop calling the atomic version in the context of a new transaction.

4.4 Optimizations

During the instrumentation, we perform several optimizations to improve the performance of DEUCE. First, we do not instrument accesses to final fields as they cannot be modified after creation. This optimization, together with the declaration of final fields whenever possible in the application code, dramatically reduces the overhead.

Second, fields accessed as part of the constructor are ignored as they are not accessible by concurrent threads until the constructor returns.

Third, instead of generating accessor methods, DEUCE actually inlines the code of the getters and setters directly in the transactional code. We have observed a slight performance improvement from this optimization.

Fourth, we chose to use the `sun.misc.Unsafe` pseudo-standard internal library to implement fast reflection, as it proved to be vastly more efficient than the standard Java reflection mechanisms. The implementation using `sun.misc.Unsafe` even outperformed the approach taken in AtomJava [12], which is based on using an anonymous class per field to replace reflection.

Finally, we tried to limit as much as possible the stress on the garbage collector, notably by using object pools when keeping track of accessed fields (read and write sets) in threads. In order to avoid any contention on the pool, we had each thread keep a separate object pool as part of its context.

Together, the above optimizations helped to significantly decrease the implementation overhead, in some of our benchmarks this improved performance by almost an order of magnitude (i.e., tenfold faster) as compared to our initial implementation.

5 Customizing Concurrency Control

DEUCE was designed to provide a research platform for STM algorithms. In order to provide a simple API for researchers to plug in their own STM algorithm's implementation, DEUCE defines the `Context` API as shown in Listing 8. The API includes an `init` method, called once before the transaction starts and then upon each retry, allowing the transaction to initialize its internal data structures. The `atomicBlockId` argument allows the transaction to log information about the specific atomic block (statically assigned in the bytecode).

```
1  public interface Context {
2    void init(int atomicBlockId);
3    boolean commit();
4    void rollback();
5
6    void beforeReadAccess(Object obj, long field);
7
8    Object onReadAccess(Object obj, Object value, long field);
9    int    onReadAccess(Object obj, int    value, long field);
10   long   onReadAccess(Object obj, long value, long field);
11   // ...
12
13   void onWriteAccess(Object obj, Object value, long field);
14   void onWriteAccess(Object obj, int    value, long field);
15   void onWriteAccess(Object obj, long value, long field);
16   // ...
17 }
```

Fig. 8. Context interface.

One of the heuristics we added to the LSA implementation is, following [6], that each one of the atomic blocks will initially be a read-only block. It will be converted to become a writable block upon retry, once it encounters a first write. Using this method, read-only blocks can save most of the overhead of logging the fields' access.

Another heuristic is that `commit` is called in case the atomic block finishes without a `TransactionException` and can return `false` in case the commit fails, which in

turn will cause a retry. A `rollback` is called when a `TransactionException` is thrown during the atomic block (this can be used by the business logic to force a retry).

The rest of the methods are called upon field access: a field read event will trigger a `beforeReadAccess` event followed by a `onReadAccess` event. A write field event will trigger an `onWriteAccess` event. DEUCE currently includes three `Context` implementations, `tl2.Context`, `lsa.Context`, and `norec.Context`, implementing the TL2 [6], LSA [19], and NOrec [5] STM algorithms. DEUCE also provides a reference implementation based on a single global lock. Since a global lock doesn't log any field access, it doesn't implement the Context interface and doesn't impose any overhead on the fields' access.

TL2 Context. The TL2 context is a straightforward implementation of the TL2 algorithm [6]. The general principle is as follows (many details omitted).

TL2 uses a shared array of *revokable versioned locks*, with each object field being mapped to a single lock. Each lock has a version that corresponds to the commit timestamp of the transaction that last updated some field protected by this lock. Timestamps are acquired upon commit from a global time base, implemented as a simple counter, updated as infrequently as possible by writing transactions.

When reading some data, a transaction checks that the associated timestamp is valid, i.e., not more recent than the time when the transaction started, and keeps track of the accessed location in its read set. Upon write, the transaction buffers the update in its write set.

At commit time, the transaction acquires (using an atomic compare-and-set operation) the locks protecting all written fields, verifies that all entries in the read set are still valid, acquires a unique commit timestamp, writes the modified fields to memory, and releases the locks. If locks cannot be acquires or validation fails, the transaction aborts, discarding buffered updates.

LSA Context. The LSA context uses the LSA algorithm [19] that was developed concurrently with TL2 and is based on a similar design. The main differences are that (1) LSA acquires locks as fields are written (encounter order), instead of at commit time, and (2) it performs "incremental validation" to avoid aborting transactions that read data that has been modified more recently than the start time of the transaction.

Both TL2 and LSA take a simple approach to conflict management: they simply abort and restart (possibly after a delay) the transaction that encounters the conflict. This strategy is simple and efficient to implement, because transactions unilaterally an abort without any synchronization with others. However, this can sometimes hamper progress by producing livelocks. DEUCE also supports variants of the TL2 and LSA algorithms that provide modular contention management strategies (as first proposed in [10]), at the price of some additional synchronization overhead at runtime.

NOrec Context. The NOrec context implements the lightweight algorithm proposed in [5]. Roughly speaking, NOrec differs from TL2 and LSA in that it uses a single ownership record (a global versioned lock) and relies on value-based validation in addition to time-based validation. The rationale of this design is to avoid having

to pay the runtime overheads associated with accessing multiple ownership records while still providing a reasonable level of concurrency thanks value-based validation which helps identify false conflicts. This algorithm was shown to be efficient at low thread counts, but tends to fall apart as concurrency increases. In contrast, TL2 and LSA provide better scalability, performing better as concurrent threads are added or when there are more frequent but disjoint commits of software transactions.

6 Java language extensions

DEUCE uses annotation to indicate transaction demarcation. Therefore, it only supports transactions at the granularity of whole methods, and cannot declare shorter "atomic blocks" as in Listing 9. In this example, one wants to transfer the balance of a set of accounts to another account. Transfer operations must be atomic, but using a single transaction for all transfers would produce an unnecessarily long transaction, and would increase the risk of conflicts with concurrent operations. In contrast, using one transaction per transfer would reduce the likelihood of aborts, and allow for better concurrency. Note that the local variable `total`, which holds the total amount transferred, must be accessed transactionally, with its previous value restored upon transactional abort.

```
1  public int transferAll(Account[] src, Account dst) {
2    int total = 0;
3    for (Account acc : src) {
4      atomic {
5        int amount = acc.balance();
6        acc.withdraw(amount);
7        dst.deposit(amount);
8        total += amount;
9      }
10   }
11   return total;
12 }
```

Fig. 9. Transactions at the granularity of atomic blocks.

Introducing new keywords to support transactional memory at the language level does not only provide finer transaction granularity, but also allows for more sophisticated constructs to control recovery, retries, etc. This allows transactions to be used in more powerful ways. Such transactional extensions do, however, require either using a pre-processor, or modifying the Java compiler to support the new constructs.

We have therefore developed a front-end tool, TMJAVA, that transforms the source code from a program written with the language extension for atomic blocks, into a program that is suitable for instrumentation by DEUCE.

The transformation performed by the front-end tool maps the atomic blocks in the extended language into annotated atomic methods. In other words, the front-end tool analyzes the code to find the atomic blocks (`atomic` keyword) inside class methods; it then creates new methods whose bodies consist of the content of the atomic blocks, and replaces the blocks with calls to these new methods. However, mapping an atomic block into an annotated atomic method is not trivial as we need to take into account several issues, notably:

1. Variables and objects, that are accessible inside the scope of the method in which the atomic block is declared, should also be available inside the scope of the atomic block.
2. Modifications inside an atomic block of variables and objects that are defined outside the scope of the block, should be visible outside the atomic block.

Variables and objects that are defined outside the atomic block, but are used and/or modified inside the atomic block, are passed as parameters to the corresponding annotated atomic method. As Java only supports parameter passing by-value for primitive types, variables that are modified inside the atomic block are passed in arrays, and updated values are copied back to the variables when returning from the atomic method (in case a single variable is modified, its new value is simply returned by the method). Listing 10 shows the DEUCE-compatible code produced by TMJAVA from Listing 9.

```
1  public class Bank {
2    public int transferAll(Account[] src, Account dst) {
3      int total = 0;
4      for (Account acc : src) {
5        total = transferAll_ab1(total, dst, acc);
6      }
7      return total;
8    }
9    @Atomic
10   private final int transferAll_ab1(int total, Account dst, Account acc) {
11     int amount = acc.balance();
12     acc.withdraw(amount);
13     dst.deposit(amount);
14     total += amount;
15     return total;
16   }
17 }
```

Fig. 10. Code of Listing 9 transformed by TMJAVA for DEUCE.

In addition to supporting basic atomic blocks, TMJAVA provides several additional keywords, notably `retry`, `next`, `leave` (for control flow), `either`/`or`/`otherwise` (for alternatives), and `on failure` (for atomic exception handling).

The TMJAVA front-end tool is based on the *JastAdd* extensible Java compiler [7]. In a first step, an abstract syntax tree (AST) is constructed from the source core. The AST is then analyzed to locate transactional language extensions, and is modified accordingly. Finally, DEUCE-compatible annotated source code is generated from the modified AST. Therefore, TMJAVA transforms Java code with TM extensions into "pure" Java source code that can be compiled using a standard compiler, but with the structure and annotations expected by DEUCE for transactifying the application.

7 Performance Evaluation

We evaluated the performance of DEUCE on a Sun UltraSPARC^{TM} T2 Plus multicore machine (2 CPUs each with 8 cores at 1.2 GHz, each with 8 hardware threads) and an Azul Vega2^{TM} (2 CPUs each with 48 cores).

7.1 DEUCE overheads

We first briefly discuss the overheads introduced by the DEUCE agent when instrumenting Java classes. Table 1 shows the memory footprint and the processing overhead when processing compiled libraries: `rt.jar` is the runtime library containing all Java built-in classes for JDK 6; `cache4j.jar` (version 0.4) is a widely used in-memory cache for Java objects; *JavaGrande* (version 1.0) is a well-known collection of Java benchmarks. Instrumentation times were measured on an Intel Core 2 Duo^{TM} CPU running at 1.8 GHz. Note that instrumentation was executed serially, i.e., without exploiting multiple cores.

Application	**Memory**		**Instrumentation**
	Original	Instrumented	**Time**
`rt.jar`	50 MB	115 MB	29 s
`cache4j.jar`	248 KB	473 KB	<1 s
JavaGrande	360 KB	679 KB	<1 s

Table 1. Memory footprint and processing overhead.

As expected, the size of the code approximately doubles because DEUCE duplicates each method, and the processing overhead is proportional to the size of the code. However, because Java uses lazy class loading, only the classes that are actually *used* in an application will be instrumented at load time. Therefore, the overheads of DEUCE are negligible considering individual classes are loaded on demand.

In terms of execution time, instrumented classes, that are not invoked within a transaction, incur no performance penalty as the original method executes.

7.2 Benchmarks

We tested the four built-in DEUCE STM options: LSA, TL2, NOrec, and the simple global lock. Our LSA version captures many of the properties of the LSA-STM [20] framework. The TL2 [6] form provides an alternative STM implementation that acquires locks at commit time using a write set, as opposed to the encounter order locking approach used in LSA.

Our experiments included three classical micro-benchmarks: a red/black tree, a sorted linked list, and a skip list, on which we performed concurrent insert, delete, and search operations. We controlled the size of the data structures, which remained constant for the duration of the experiments, and the mix of operations.

We also experimented with three real-world applications, the first implements the Lee routing algorithm (as presented in Lee-TM [2]). It is worth noting that adapting the application to DEUCE was straightforward, with little more than a change of synchronized blocks into atomic methods. The second application is called Cache4J, an LRU cache implementation. Again adapting the application to DEUCE was straightforward. The third application comes from the widely used STAMP [13] TM benchmark suite.

Micro-benchmarks. We began by comparing DEUCE to DSTM2, the only competing STM framework that does not require compiler support. We benchmarked DSTM2 and DEUCE based on the Red/Black tree benchmark provided with the DSTM2 framework. We tried many variations of operation combinations and levels of concurrency on both the Azul and Sun machines. Comparison results for a tree with 10k elements are shown in Figure 11, we found that in all the benchmarks DSTM2 was about 100 times (two orders of magnitude) slower than DEUCE. Our results are consistent with those published in [9], where DSTM2's throughput, measured in operations per second, is in the thousands, while DEUCE's throughput is in the millions. Based on these experiments, we believe one can conclude that DEUCE is the first viable compiler and JVM independent Java STM framework.

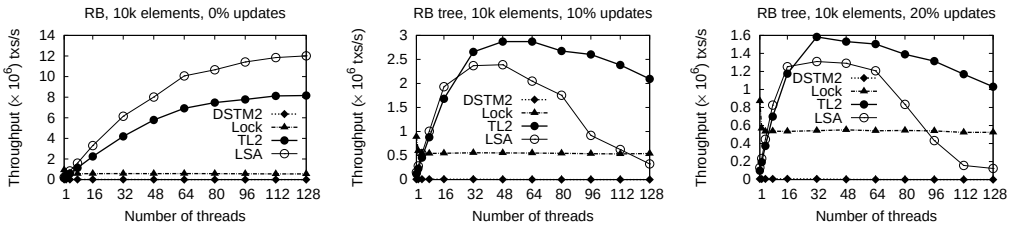

Fig. 11. The red/black tree benchmark (Sun).

On the other hand we observe that the DEUCE STMs scale in an impressive way even with 20% updates (up to 32 threads). While DSTM2 shows no scalability. When we investigated deeper we found out that most of DSTM2 overhead rest in two areas. The most significant area is the contention manager which acts as a contention point, the second area is the reflection mechanism used by DSTM2 to retrieve and assign values during the transaction and in commit.

Next we present the results of the tests with the linked list and the skip list on the Sun and Azul machines. The linked list benchmark is known to produce a large degree of contention as all threads must traverse the same long sequence of nodes to reach the target element. Results of experiments are shown in Figure 12 and Figure 13 for a linked list with 256 elements and different update rates. We observe that because there is little potential for parallelism, the single lock, which performs significantly less lock acquisition and tracking work than the STMs, wins in all three benchmarks. The STMs scale slightly until 30 threads when there are up to 5% updates, and then drop due to a rise in overhead with no benefit in parallelism. With 20% updates, the max is reached at about 5 threads, because as concurrency further increases one can expect at least 2 concurrent updates, and the STMs suffer from repeated transactional aborts.

Finally, we consider results from the skip list benchmark. Skip lists [18] allow significantly greater potential for parallelism than linked lists because different threads are likely to traverse the data structure following distinct paths. Results for a list with 16k elements are shown in Figure 14 and Figure 15. We observe that the STMs scale in an impressive way, as long as there is an increase in the level of parallelism, and provide great scalability even with 20% updates as long as the abort rate remains

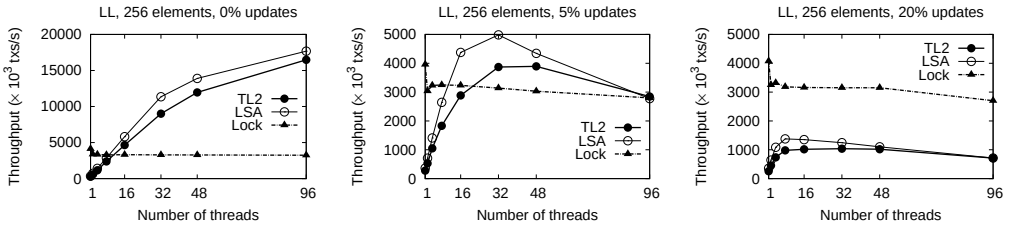

Fig. 12. The linked list benchmark (Sun).

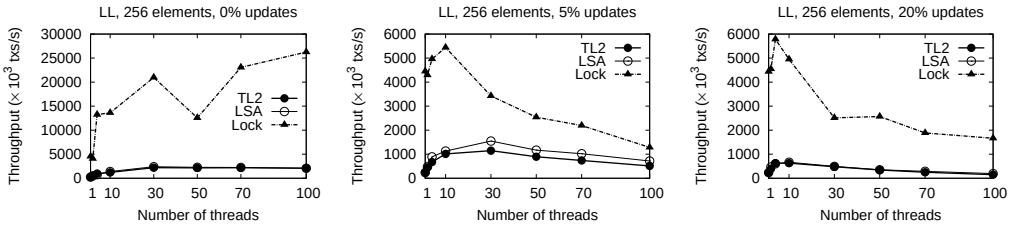

Fig. 13. The linked list benchmark (Azul).

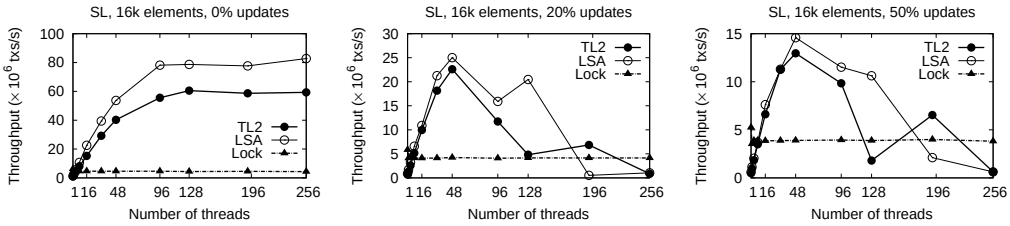

Fig. 14. The skiplist benchmark (Sun).

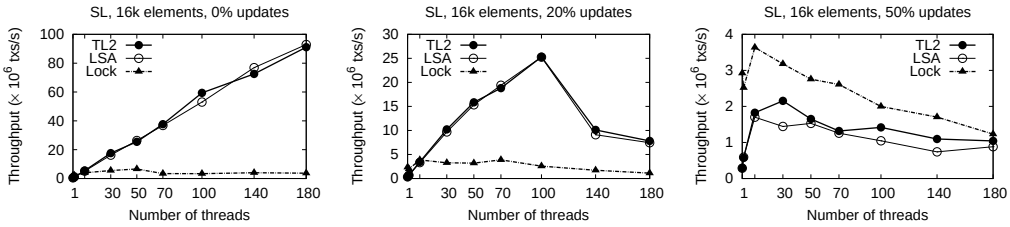

Fig. 15. The skiplist benchmark (Azul).

at a reasonable level. We added a benchmark with 50% updates to show that there is a limit to their scalability. When we increase the fraction of updates to 50%, the STMs in Figure 15 reach a "melting point" much earlier (at about 10 threads versus 100 threads in the 20% benchmark) and overall the lock wins again because the abort rate is high and the STMs incur and overhead without a gain in parallelism. We note that typically search structures have about 10% updates.

Real applications. Our last benchmarks demonstrate how simple it is to replace the critical sections with transactions. The first takes a serial version of the Lee rout-

ing algorithm and demonstrates how a simple replacement of the critical sections by transactions significantly improves scalability. The second takes a non-multi-threaded lock based a LRU cache implementation (Cache4J) and shows that it is straightforward to replace the critical sections, but scalability isn't promised.

Circuit routing is the process of automatically producing an interconnection between electronic components. Lee's routing algorithm is an attractive candidate for parallelization since circuits (as shown in [2]) consist of thousands of routes, each of which can potentially be routed concurrently.

The graph in Figure 16 shows execution time (that is, latency, not throughput) for three different boards with the TL2, LSA, and NOrec algorithms. As can be seen, DEUCE scales well with all algorithms, with the overall time decreasing even in the case of a large board (MemBoard). NOrec exhibits better performance with a single thread due to the lower overhead of its single ownership record. With more threads, the performance of all algorithms is virtually identical. This can be explained by the natural parallelism and low contention of this benchmark.

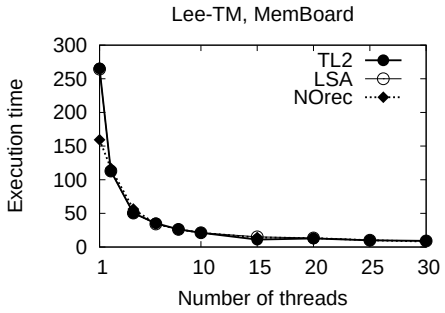

Fig. 16. The Lee Routing benchmark (Sun).

Cache4J is an LRU lock-based cache implementation. The implementation is based on two internal data structures, a tree and a hash-map. The tree manages the LRU while the hash-map holds the data. The Cache4J implementation is based on a single global lock which naturally leads to no scalability. The graph in Figure 17 shows the result of replacing the single lock with transactions using the LSA algorithm. As can be seen, DEUCE doesn't scale well, with the overall throughput slightly decreasing. A quick profiling shows that the fact that Cache4J is an LRU cache implies that every get operation also updated the internal tree. This alone makes every transaction an update transaction. Yet, the fact that the total throughput remains almost the same even with 80 threads is encouraging due to the fact that transactional memory's main advantages, besides scalability, are code simplicity and robustness.

Finally, we tested DEUCE on the Vacation benchmark from STAMP [13], which is the most widely used TM benchmark suite. As STAMP was been originally written in C, we used a Java port of the original benchmarks adapted for DEUCE. The Vacation application models an online travel reservation system, implemented as a set of trees that keep track of customers and their reservations. Threads spend time executing

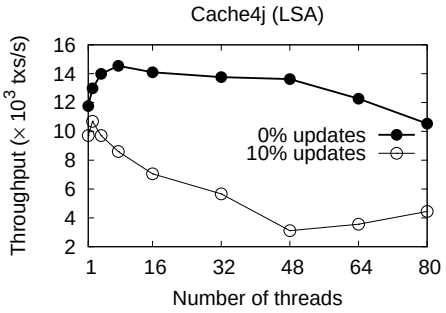

Fig. 17. The Cache4J benchmark (Sun).

transactions, and the contention level is moderate. Figure 18 shows the execution time of Vacation with the TL2, LSA, and NOrec algorithms. We can observe that both TL2 and LSA scale well with the number of cores. In contrast, NOrec shows almost no scalability, although it performs better than TL2 and LSA under low thread counts. This can be explained by the fact that (1) NOrec's single ownership record is penalized by the size of the transactions, and (2) value-based validation is quite expensive in Java since it requires accessing memory via APIs that are slower than the direct memory accesses provided by unmanaged languages.

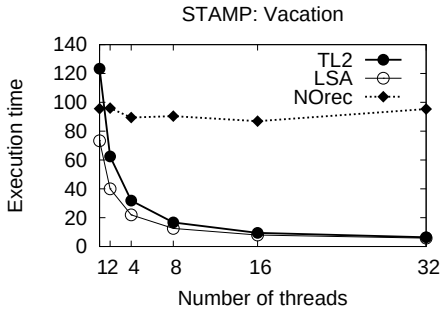

Fig. 18. The Vacation benchmark from the STAMP suite (Sun).

In conclusion, DEUCE shows the typical performance patterns of an STM, good scalability when there is a potential for parallelism, and unimpressive performance when parallelism is low. The results, however, are very encouraging considering the fact that DEUCE, a pure Java library with all the implied overheads, shows scalability even at low thread counts.

8 Conclusion

We introduced DEUCE, a novel open-source Java framework for transactional memory. As we showed, DEUCE is the first efficient fully featured Java STM framework

that can be added to an existing application without changes to its compiler or libraries. It demonstrates that one can design an efficient, pure Java STM without a compiler support, even though language support as provided by the companion front-end TMJava provides additional expressiveness.

Though much work obviously remains to be done, we believe Deuce is ready for use by developers. It is freely downloadable from `http://code.google.com/p/deuce` under the Apache license.

In conclusion, Table 2 shows a general overview comparing the two well-known Java STMs AtomJava [1] and DTMS2 vs. Deuce. This comparison shows that Deuce ranks highest overall, and provides novel capabilities which yield much better usability and performance than formerly existed.

	AtomJava	DSTM2	Deuce
`Locks`	Object based	Object based	Field based
`Instrumentation`	Source	Runtime	Runtime
`API`	Non-Intrusive	Intrusive	Non-Intrusive
`Libraries support`	None	None	Runtime & offline
`Fields access`	Anonymous classes	Reflection	Low level `unsafe`
`Execution overhead`	Medium	High	Medium
`Extensible`	Yes	Yes	Yes
`Transaction context`	`atomic` block	`Callable` object	`@Atomic` annotation

Table 2. Comparing AtomJava, DSTM2, and Deuce

Acknowledgements. This paper was supported in part by grants from Sun Microsystems, Intel Corporation, Microsoft Inc., as well as a grant 06/1344 from the Israeli Science Foundation, European Union grant FP7-ICT-2007-1 (project VELOX), and Swiss National Foundation grant 200021-118043. The TMJava front-end has been developed by Derin Harmanci.

References

1. B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The jikes research virtual machine project: building an open-source research community. *IBM Syst. J.*, 44(2):399–417, 2005.
2. M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. Luján, and K. Jarvis. Lee-tm: A non-trivial benchmark suite for transactional memory. In *Proc. of the International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, pages 196–207, Berlin, Heidelberg, 2008. Springer-Verlag.
3. W. Binder, J. Hulaas, and P. Moret. Advanced java bytecode instrumentation. In *Proc. of the International Symposium on Principles and Practice of Programming in Java (PPPJ)*, pages 135–144, New York, NY, USA, 2007. ACM.
4. B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The atomos transactional programming language. *SIGPLAN Not.*, 41(6):1–13, 2006.

[1] We did not show comparison benchmark results with AtomJava due technical limitations and bugs found in its current implementation, but we have observed on small benchmarks that Deuce outperforms AtomJava by a comfortable margin.

5. L. Dalessandro, M. Spear, and M. Scott. Norec: streamlining stm by abolishing ownership records. In *Proc. of the ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 67–78, Jan 2010.
6. D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proc. of the International Symposium on Distributed Computing (DISC)*, pages 194–208, 2006.
7. T. Ekman and G. Hedin. The JastAdd extensible Java compiler. *SIGPLAN Not.*, 42(10):1–18, 2007.
8. T. Harris and K. Fraser. Language support for lightweight transactions. In *Proc. of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 388–402, New York, NY, USA, 2003. ACM.
9. M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *Proc. of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 253–262, New York, NY, USA, 2006. ACM.
10. M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proc. of the Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, New York, NY, USA, 2003. ACM.
11. M. Herlihy and E. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.
12. B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *Proc. of the Workshop on Memory System Performance and Correctness (MSPC)*, pages 82–91, New York, NY, USA, 2006. ACM.
13. C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of IISWC*, September 2008.
14. Multiverse. http://multiverse.codehaus.org/.
15. Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for C/C++. In *Proc. of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 195–212, New York, NY, USA, 2008. ACM.
16. N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In *Proc. of the International Conference on Compiler Construction (CC)*, pages 138–152, 2003.
17. W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.
18. W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *ACM Transactions on Database Systems*, 33(6):668–676, 1990.
19. T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proc. of the International Symposium on Distributed Computing (DISC)*, pages 284–298, Sep 2006.
20. T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *Proc. of the Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 221–228, New York, NY, USA, 2007. ACM.
21. N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, February 1997.
22. L. Ziarek, A. Welc, A.-R. Adl-Tabatabai, V. Menon, T. Shpeisman, and S. Jagannathan. A uniform transactional execution environment for java. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 129–154, Berlin, Heidelberg, 2008. Springer-Verlag.