A Uniform Transactional Execution Environment for Java

Lukasz Ziarek[1], Adam Welc[2], Ali-Reza Adl-Tabatabai[2], Vijay Menon[2], Tatiana Shpeisman[2], and Suresh Jagannathan[1]

[1]Dept. of Computer Sciences
Purdue University
West Lafayette, IN 47907

[2]Programming Systems Lab
Intel Corporation
Santa Clara, CA 95054

{lziarek,suresh}@cs.purdue.edu
{adam.welc,ali-reza.adl-tabatabai,vijay.s.menon,tatiana.shpeisman}@intel.com

Abstract. Transactional memory (TM) has recently emerged as an effective tool for extracting fine-grain parallelism from declarative critical sections. In order to make STM systems practical, significant effort has been made to integrate transactions into existing programming languages. Unfortunately, existing approaches fail to provide a simple implementation that permits lock-based and transaction-based abstractions to coexist seamlessly. Because of the fundamental semantic differences between locks and transactions, legacy applications or libraries written using locks can not be transparently used within atomic regions. To address these shortcomings, we implement a uniform transactional execution environment for Java programs in which transactions can be integrated with more traditional concurrency control constructs. Programmers can run arbitrary programs that utilize traditional mutual-exclusion-based programming techniques, execute new programs written with explicit transactional constructs, and freely combine abstractions that use both coding styles.

1 Introduction

Over the last decade, transactional memory (TM) has emerged as an attractive alternative to lock-based abstractions by providing stronger semantic guarantees (atomicity and isolation) as well as a simpler programming model. Transactional memory relieves the burden of reasoning about deadlocks and locking protocols. Additionally, transactional memory has also been utilized to extract fine-grain parallelism from declarative critical sections. In particular, software transactional memory (STM) systems provide scalable performance surpassing that of coarse-grain locks and a simpler, but competitive alternative to hand-tuned fine-grain locks [3, 10, 12, 25, 30].

In order to make STM systems practical, significant effort has been made to integrate transactions into existing programming languages, virtual machines, and run-time systems. Since languages such as Java already provide concurrency

control primitives based on mutual exclusion, seamlessly integrating transactions into these languages requires rectifying the semantics and implementation of the two constructs. Existing approaches that attempt to support different concurrency control mechanisms [10, 30] do not provide a uniform implementation. Therefore, complex programs that make use of mutual-exclusion cannot be executed on a system providing transactional support without breaking composability and abstraction – to be assured that it is safe to execute a code region transactionally requires knowing that methods invoked within the dynamic context of this region do not make use of mechanisms that violate transactional semantics such as I/O or communication. Such disparities between different concurrency control mechanisms have prevented the integration of transactions into large, complex programs and limited mainstream deployment. As a result, programmers cannot easily compose transactions with other concurrency primitives.

In this paper, we describe a uniform transactional execution environment for Java programs in which transactions and other concurrency control constructs can be seamlessly integrated, interchanged, and composed. Programmers can run arbitrary programs that utilize traditional mutual-exclusion-based programming techniques, execute new programs written with explicit transactional constructs, and freely combine abstractions that use both coding styles. Our framework allows programmers to write modular transactional code without having to hand-inspect all calls within a transactional region to guarantee safety. The uniform transactional execution environment is composed of two mutually co-operating implementations, one for locks and the other for transactions, and allows for dynamic handoff between different concurrency control primitives.

Our paper makes the following contributions:

1. We describe how explicit transactional constructs can be seamlessly integrated into Java. We present a programming model which provides both *synchronized* and *atomic* primitives, and a uniform semantics for composing and interchanging the two.
2. We provide an in-depth exploration of how transactions can be used to support execution of lock-based synchronization constructs. Our study includes issues related to language memory models, and concurrency operators which inherently break isolation such as `wait` and `notify`. We explore properties that must be satisfied by a transactional implementation striving to address these issues. We present the theoretical foundations of such an implementation we call P-SLE (*pure-software lock elision*).
3. We present the design and implementation of the first fully uniform execution environment supporting both traditional (locks) and new (atomic blocks) constructs.
4. We evaluate the performance of our system on a set of non-trivial benchmarks demonstrating scalability comparable to programs using fine-grained mutual-exclusion locks, and improved performance over coarse-grain locks. Our benchmarks perform I/O actions, inter-thread communication, thread spawns, class loading, and reflection, exercising a significant portion of Java language features.

T1	T2
`synchronized(hmap1) {`	
`  synchronized(hmap2) {`	
	`synchronized(hmap1) {`
`    hmap2.move(hmap1);`	
	`  hmap1.get(x);`
`  }`	`}`
`}`	

Fig. 1. An example program where locks can be elided, allowing for additional concurrency.

2 Motivation

Locks are the most pervasive concurrency control mechanism used to guard shared data accesses within critical sections. Their semantics is defined by the mutual exclusion of critical sections. In addition to providing *thread synchronization* and preventing interference between locked regions, lock operations act as *memory synchronization* points providing ordering and visibility guarantees [16]. Software transactional memory, advocated as a lock replacement mechanism, unfortunately provides different semantics. STMs guarantee *atomicity* and *isolation* of operations executed within critical sections (also known as *atomic regions* or *atomic blocks*) to prevent interference. The exact semantics provided by an STM is defined in terms of an underlying transactional implementation. For example, STMs with weakly atomic and strongly atomic semantics [4, 26] are implemented differently. STM systems also typically define their own notions of ordering and visibility (e.g., closed vs. open nesting [21]). Due to differences in both semantics and implementations, locks and transactions cannot easily be composed or interchanged. For example, mutual exclusion may hinder extraction of additional scalability, whereas semantic properties of atomic regions may violate visibility guarantees provided by lock-based synchronized blocks.

To provide composability, differences in semantics and implementations must be rectified. The semantics of locked regions and atomic blocks must be consistent and their implementations uniform. Observe that the semantics of locks may be supported by an implementation different from mutual exclusion and, similarly, alternative implementations can be used to implement the semantics dictated by transactions. The following examples illustrate some of the issues that arise in defining a uniform implementation and consistent semantics for both constructs.

Consider the example Java program in Figure 1 which consists of two threads operating over two different hashmaps `hmap1` and `hmap2`. Although highly concurrent lock-based Java implementations of hashmap exist, exclusively locking the hashmap object to perform operations on the hashmap as a whole fundamentally limits scalability. In the example in Figure 1 thread `T1` moves the content of hashmap `hmap1` into `hmap2`, locking both hashmaps and thus preventing thread

```
synchronized(m) {
  count--;
  if (count == 0) m.notifyAll();
  while (count != 0) {
    m.wait();
  }
}
```

Fig. 2. Barrier Example

T2 from concurrently accessing hmap1. In some cases, as seen in the example in Figure 1, locks can be elided – their implementation, mutual exclusion of critical sections, can be replaced by a transactional one. This can be accomplished by rewriting source code to utilize transactions without changing the semantics of the original program. If locks were elided, thread T2 in Figure 1 would be able to perform its operations concurrently with thread T1. The underlying transactional machinery would guarantee correctness and consistency of the operations on hmap1 and hmap2. To summarize, in this example either transactions or locks may be utilized with no change to the underlying program semantics.

Lock elision, however, is not always possible. Consider an example program that spawns multiple worker threads that perform work over a collection of shared structures. Data computed by those threads is then collected and aggregated (SPECjbb2000 [29] is an example of such a program). Such programs use coordination barriers to synchronize the worker threads so that data may be compiled. Coordination barriers typically use a communication protocol that allows threads to exchange information about their arrival at the barrier point. Consider a simple counter-based Java implementation that notifies all threads waiting on the barrier when the counter (initialized to the total number of threads) reaches zero (Figure 2).

A naive translation of the synchronized block in Figure 2 to use transactions is problematic for multiple reasons. First, the execution of the wait and notify methods inside of atomic regions is typically prohibited by STMs [3, 10]. Second, even if an STM defined meaningful behavior for the invocation of such methods inside atomic regions, the execution of the barrier would not complete. The update to the internal counter would never become visible because transactions impose isolation requirements on the code they protect.

A potential solution to translating the example in Figure 2 to use atomic regions must therefore not only support wait/notify but also allow updates to the internal counter to become visible to other threads. One solution, suggested by [27], is to expose the value of the internal counter by explicitly violating isolation of the original atomic region – splitting the atomic region into multiple separate regions without altering the behavior of the program. Isolation would also have to be broken to support wait/notify primitives. Breaking isolation in such a manner creates a race condition on accesses to the shared counter because it is no longer protected within the same contiguous critical region. Alternatively, we

T1	T2
`atomic {`	
`  synchronized(m) {`	
	`synchronized(m) {`
`    foo();`	
`  }`	`  bar();`
`  ...`	`}`
`}`	

Fig. 3. Composition of synchronized blocks and atomic regions

can leave the synchronized block unmodified. Such a solution requires reasoning about all possible interactions between the synchronized blocks and the atomic regions present in the program.

Although it may be possible to translate the example code in Figure 2 with extensions to an STM, previous work [4] suggests that even with access to all source code a translation of synchronized blocks to atomic regions that retains the original program's semantics is not always feasible. At best, such a translation requires careful analysis of the original program's source code. However, source code may not always be available, might be complex, and may need to be re-engineered for transactions. Our own experience in translating lock-based versions of some well-known benchmarks into their transactional equivalents [26] mirrors these findings. Even with STM extensions, it is still unclear if certain synchronized blocks can even be translated at all, motivating the necessity of supporting composability between synchronized blocks and atomic regions.

Unfortunately, composability of different concurrency control primitives is not easily achieved. Since atomic regions and synchronized blocks provide different semantic guarantees on visibility and isolation, composition of the two constructs can yield non-intuitive and potentially erroneous behavior. Consider the program in Figure 3. In the common case, locks protecting synchronized blocks in both threads might be elided allowing execution to proceed fully under transactional control. However, consider a situation when method `foo()` executes a native call and prints a message to the screen. In order to prevent the message from being printed more than once, a possibility that could arise if the synchronized block is implemented transactionally, the block must be expressed using mutual exclusion semantics. Additionally, once thread `T1` acquires lock `m`, the synchronized block executed by thread `T2` must also be implemented using mutual exclusion. Moreover, an abort may still be triggered after thread `T1` finishes executing its synchronized block but before it finishes executing the outer atomic region. As a result, the entire atomic region in thread `T1` must also be prevented from aborting, and thus must be implemented using mutual exclusion.

To address these issues, this paper presents a uniform execution environment that allows *safe* interoperability between lock-based and transaction-based concurrency primitives. The execution environment *dynamically* switches between transactional and lock based implementations to provide consistent semantics for

both constructs. This allows for arbitrary composition and interchangeability of synchronized blocks and atomic regions.

3 Programming and Execution Model

Our system supports concurrent programming in Java by offering two basic primitives to programmers: `synchronized` providing lock semantics and `atomic` providing transactional semantics. The system imposes no constraints on how the two primitives may interact. Programmers may utilize both primitives for concurrency control and can compose them arbitrarily. Additionally, there are no restrictions on what code can be executed within the dynamic context of a transaction, allowing support for I/O, legacy and native code.

Both primitives are realized using a transactional implementation. In our system, synchronized blocks are automatically and transparently transformed to implicitly use a transactional implementation we call pure-software lock elision (P-SLE), rather than an implementation based on mutual exclusion. We explore properties that the P-SLE implementation must satisfy in order to preserve lock semantics in Section 4. User-defined atomic regions are already implemented using transactions and thus require no additional support under P-SLE (details are given in Section 5).

Since transactions and locks differ in their semantic definitions, execution of lock-based critical sections using P-SLE may not always be possible. When such situations are detected, our system seamlessly reverts critical sections to use an implementation based on mutual exclusion. This is accomplished through a *fallback* mechanism discussed in Section 4. Fallback has a natural definition for transactions translated from critical sections: acquire the original lock as defined by the input program. User injected atomic regions, however, are not defined with regard to locks. Thus, we present a new mechanism called *atomic serialization* which allows for the execution of user-defined transactions under mutual exclusion. Conceptually, atomic serialization works by effectively serializing execution of atomic regions using a single global lock. Atomic serialization aborts a transaction which must fallback and acquires a global lock prior to re-executing the critical region protected by the user-defined transaction.

Our system thus optimistically executes all concurrency control primitives transactionally. In situations where such an execution is infeasible (e.g., eliding locks violates lock semantics), the implementation switches back to using mutual exclusion. This transition is one-directional – once execution of a critical section reverts to using mutual exclusion, it will run in this execution mode until completion. Figure 4 illustrates the system depicting both concurrency control primitives and the transitions supported by the implementation.

4 Pure-Software Lock Elision (P-SLE)

Our first step to constructing a uniform transactional execution environment for Java programs is replacing the existing implementation for Java's synchro-

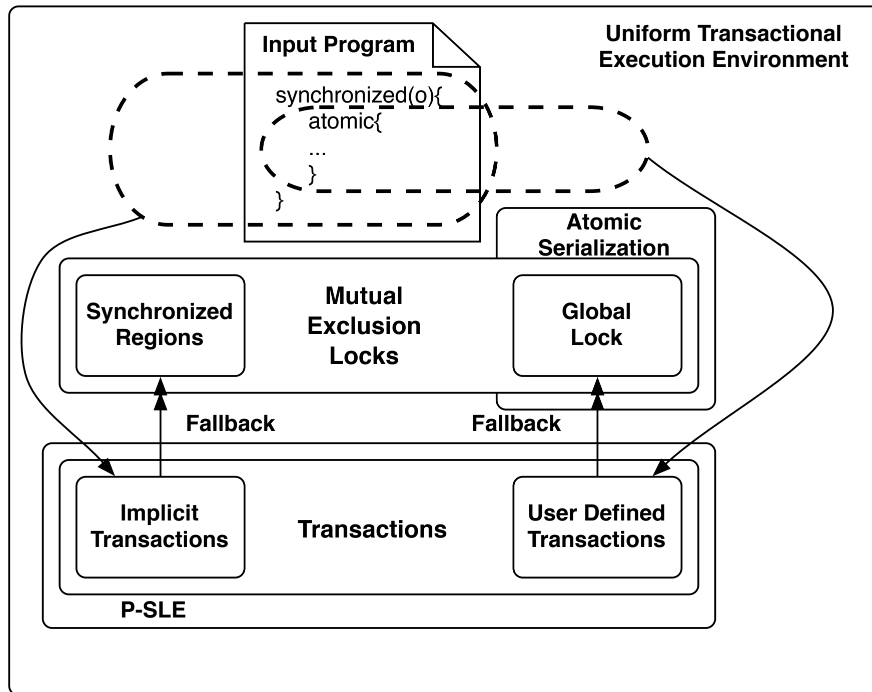


Fig. 4. Execution model for a program containing both synchronized blocks and atomic regions. The uniform execution environment utilizes both a lock-based implementation and a transactional one. Double arrows represent the fallback mechanism, while single arrows show the initial implementation underlying both concurrency control primitives.

nized blocks (i.e., mutual exclusion) with P-SLE. Because in doing so we are obligated to preserve lock semantics, P-SLE must provide properties mirroring mutual exclusion, that is, both thread and memory synchronization effects of lock operations. The JMM uses these lock properties to define valid executions for Java programs.

4.1 Correctness

Clearly, in order for one implementation to be correctly replaced by the other we must define a correlation between their semantics. We do so in terms of program schedules produces by each implementation. A program schedule reflects the execution of a concurrent program on a single-processor machine and defines a total order among program operations. This notion of a schedule can be easily generalized to a multi-processor case – operations whose order cannot be determined when analyzing the execution of a program are independent and can be executed in arbitrary order. For schedules generated under a transactional

implementation, only operations of the last successful execution of a transaction become part of the program schedule.

Our first step in defining the correctness property is to determine what it means for two schedules to be equivalent under the JMM. The JMM defines a *happens-before* relation (written $\stackrel{hb}{\rightarrow}$) among the actions performed by threads in a given execution of a program. For single-threaded executions, the happens-before relation is defined by program order. For multi-threaded executions, the happens-before relation is defined between pairs of synchronization operations, such as the begin and end of a critical section, or the write and read of the same volatile variable. The happens-before relation is transitive: $\mathtt{x}\stackrel{hb}{\rightarrow}\mathtt{y}$ and $\mathtt{y}\stackrel{hb}{\rightarrow}\mathtt{z}$ imply $\mathtt{x}\stackrel{hb}{\rightarrow}\mathtt{z}$.

The JMM uses the happens-before relation to define visibility requirements for operations in a given schedule. Consider a pair of read and write operations, $\mathtt{r}_v$ and $\mathtt{w}_v$, accessing the same variable v and ordered by the happens-before relation ($\mathtt{w}_v \stackrel{hb}{\rightarrow} \mathtt{r}_v$). Assume further that no intervening write exists such that $\mathtt{w}_v \stackrel{hb}{\rightarrow} \mathtt{w}'_v \stackrel{hb}{\rightarrow} \mathtt{r}_v$. In other words, $\mathtt{w}_v$ is the "latest" write to v preceding $\mathtt{r}_v$ in the order defined by the happens-before relation. Then, $\mathtt{r}_v$ is obligated to observe the effect of $\mathtt{w}_v$, unless intervening writes to v unordered by the happens-before relation, exist between $\mathtt{w}_v$ and $\mathtt{r}_v$. In this case, $\mathtt{r}_v$ may observe either a value produced by $\mathtt{w}_v$ or a value produced by any of the intervening writes. We say that two schedules are identical if all happens-before relations are preserved between the same operations in both schedules.

The JMM has been defined under an implicit assumption that critical sections are implemented using mutual exclusion locks. Given a program P, the JMM defines a set of possible schedules, S, that characterizes P's runtime behavior. Clearly, a transactional version of P, $\tau(P)$, cannot produce a schedule s such that $s \notin S$. Similarly, if $S = \{s\}$ then $\tau(P)$ can only have one unique schedule as defined by the JMM, namely s. Thus, a transactional version of a Java program cannot produce any new schedules and it must produce the exact schedule the original program produces if only one exists. However, what occurs when multiple schedules are plausible for a program P? The JMM itself does not enforce any scheduling fairness restrictions. The underlying virtual machine and its scheduler are free to provide any proper subset of schedules for P. We leverage the freedom provided by the JMM in defining correct executions for a transactionalized program. If program P produces a set of schedules S under the JMM and $\tau(P)$ produces a set of schedules S' then $\tau(P)$ is a correct execution if $S' \subseteq S$.

One could argue that a uniform transactional execution environment should only be obligated to provide correctness guarantees for transactional and lock-based executions when the program is *properly structured* [30]. Such programs do not exhibit races, have shared data protected by a consistent set of locks, etc. Unfortunately, requiring programs to be properly structured in order to leverage transactional execution is a severe restriction in practice and prevents

Property	Mutual Exclusion	Weak Atomicity	Strong Atomicity	P-SLE
RR	*no*	*no*	*yes*	*yes/no**
IU	*no*	*no*	*yes*	*yes/no**
IR	*no*	*yes/no**	*yes*	*yes/no**
SS	*yes*	*yes/no**	*yes*	*yes*
PUS	*yes*	*yes/no**	*yes*	*yes*
PRS	*yes*	*yes/no**	*yes*	*yes*
GS	*yes*	*yes/no**	*yes*	*yes*

Table 1. A list of safety properties for isolation and ordering concerns related to shared memory accesses (* – depends on a particular incarnation).

such programming idioms as privatization [26]. Our focus is on ensuring well-defined behavior even when programs are not properly structured.

We identify a set of properties that the P-SLE implementation must satisfy in order to correctly support lock semantics. We do so by analyzing properties of existing implementations: both non-transactional (mutual exclusion) and transactional (weak atomicity and strong atomicity [1]). Our discussion is separated into two parts: one concerning problems related to isolation and ordering of operations that may lead to incorrect results being computed, and the other concerning problems related to visibility of an operation's effects that may prevent programs from making progress. Problems related to isolation and ordering have been examined in previous work [19, 26] and our system builds off of such solutions.

4.2 Isolation and Ordering Concerns

Table 1 presents a classification of isolation and ordering safety properties preserved by different implementations (*yes* means that the implementation supports the property, *no* means that it does not and *yes/no* means that different incarnations of a particular implementation exist which may or may not preserve it [2].). In the following, accesses to shared variables can be either protected within critical sections or unprotected.

The first three properties described in the table concern direct interactions between protected and unprotected accesses. In order to provide some intuition behind their definition, Figure 5 demonstrates what happens if these properties are violated. The code samples in Figure 5 use the same shared variable `x` for illustration of the violations and (as well as all the remaining figures in this section) are written in "pseudo-Java" – we use the `critical` keyword to denote critical sections (instead of `synchronized` or `atomic`) to avoid implying a particular implementation of a concurrency control mechanism. We assume that if

[1] We assume that both weak atomicity and strong atomicity use close nesting.

[2] For example, some incarnations of weak atomicity use updates in-place while the others use write buffering

Initially x==0

T1	T2
critical	
{ r1=x;	
	x=1;
r2=x; }	

(a) r1!=r2 violates RR

Initially x==0

T1	T2
critical	
{ r=x;	
	x=10;
x=r+1; }	

(b) x==1 violates IU

Initially x==0

T1	T2
critical	
{ x++;	
	r=x;
x++; }	

(c) odd r violates IR

Fig. 5. Safety violations resulting from direct interactions between protected and unprotected accesses.

mutual exclusion is used to execute critical sections, then all threads synchronize using the same global lock.

Preservation of repeatable reads (RR) requires that protected reads of the same shared variable by one thread must return the same value despite intermediate unprotected write to the same variable by another thread being executed between the reads. The intermediate updates (IU) property is preserved if the effect of an unprotected update to some shared variable happening between a protected read of the same variable and a protected write of the same variable is not lost. Finally, preservation of the intermediate reads (IR) property requires that an unprotected read of some shared variable does not see a dirty intermediate value available between two protected writes of the same variable. Since mutual exclusion preserves none of these properties, they also do not need to be preserved by P-SLE.

The next property described in the table, speculation safety (SS)[3], concerns (possibly indirect) interactions between protected and unprotected accesses combined with the effects of speculative execution. Speculation safety prevents a protected speculative write to some variable from producing an "out-of-thin-air" value that could be observed by an unprotected read in another thread. Mutual exclusion trivially preserves speculation safety since no protected access is ever speculative under this implementation. As a result, P-SLE must preserve this property as well, but in case of transactional implementations special care needs to be taken to satisfy it – strong atomicity preserves it but weak atomicity may or may not, depending on its particular incarnation. The example in Figure 6 illustrates one possible scenario when speculation safety gets violated – under mutual exclusion, thread T2 could never observe r==1 since critical sections of threads T1 and T2 would be executed serially and thread T2 would never see values of y and z to be different from each other. When executed transactionally, the transaction executed by thread T2 could observe different values of y and z, and even though the transaction would be later aborted because of an inconsistent state, it would still perform an update to x producing a value out of thin air, visible to thread T3.

[3] This safety property subsumes another safety property discussed in previous work [19] – observable consistency

T1	T2	T3
critical		
{	critical	
y++;	{	
	if(y!=z) x=1;	
		r=x;
z++;	// abort	
}	}	

Fig. 6. `r==1` violates SS.

The following two safety properties, privatization safety (PRS) and publication safety (PUS), concern ordering of protected accesses with respect to unprotected accesses. These idioms reflect how a privatizing or publishing action can convert data from shared to private state, or vice versa. The privatization safety pattern is generalized in Figure 7(a) – some memory location is shared when accessed in `S1` but private to `T2` when accessed in `S2`. An intervening privatizing action ensures that the location is only accessible to `T2` and involves execution of a memory synchronization operation to guarantee that other threads are aware of the privatization event. The publication pattern, generalized in Figure 7(b), is a reverse of the publication pattern. Both patterns have been well-researched [19, 1, 26, 28], and the conclusion is that while mutual exclusion trivially preserves both safety properties, it is not necessarily automatically true for transactional implementations, such as P-SLE, and requires special care to provide the same memory synchronization effects.

The last property, granular safety (GS), prevents protected writes from affecting unprotected reads of adjacent data. In Java, a protected write to a field `x` of an object should have no effect on a read of field `y` of the same object. By definition, granular safety cannot be violated when using mutual exclusion locks since no protected accesses ever modify adjacent data (protected reads and writes occur at the sane granularity as "regular" data accesses). In order for the same guarantee to hold for transactional implementations, special care may have to be taken on how modified data is logged and written to shared memory.

T1	T2
critical	
{	
S1;	
}	
	[privatizing action]
	S2;

(a) privatization

T1	T2
S1;	
[publication action]	
	critical
	{
	S2;
	}

(b) publication

Fig. 7. Safety of protected vs. unprotected accesses ordering.

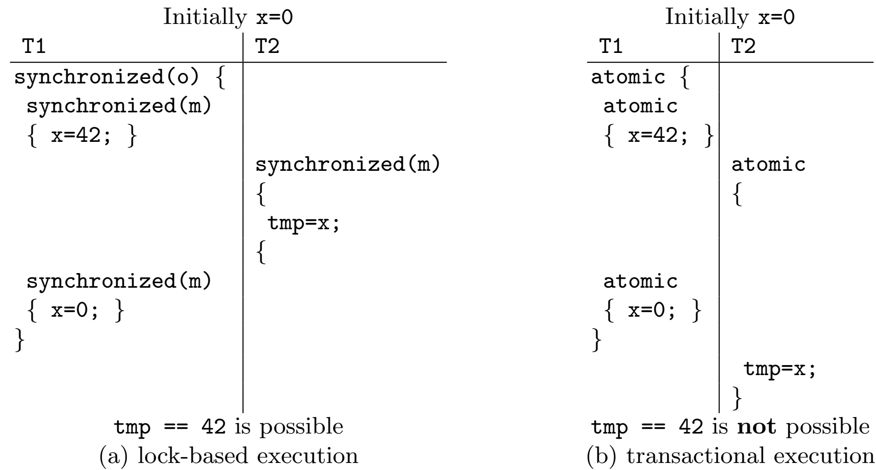


Fig. 8. Visibility in presence of inner synchronization-related operations

Thus, P-SLE must provide at least the same level of isolation and ordering guarantees as mutual exclusion provides. At the same time, according to our correctness definition presented in Section 4.1, P-SLE can provide stronger guarantees than mutual exclusion since our obligation is to reproduce only a subset of all schedules legal under the JMM. For example, an implementation of a P-SLE system can allow or disallow violation of any of the first properties listed in Table 1, provided that the visibility properties described in the next section are satisfied.

4.3 Visibility Concerns

While the isolation and ordering properties of transactional systems have recently attracted significant attention, issues concerning mismatches between visibility properties of systems supporting mutual exclusion semantics and those supporting transactional semantics have been, with some notable exceptions [4, 30], largely neglected.

Visibility issues are closely tied to progress guarantees provided by the underlying execution engine. The JMM (or, in fact, the Java Language Specification [8] or the Java Virtual Machine Specification [15]) does not require the Java execution environment to provide any guarantees with respect to application progress or scheduling fairness. As a result, a legal implementation of a JVM could attempt to execute all threads in sequential order, getting "stuck" when control dependencies among operations in these threads manifest. In other words, it is legal for a JVM to *never* successfully complete an inter-thread communication action, such as the coordination barrier presented in Section 2 in Figure 2. While we certainly agree that different JVM implementations are free to make their own scheduling decisions, we also believe that certain programs are intuitively

Property	Mutual Exclusion	Weak Atomicity	Strong Atomicity	P-SLE
SV	*yes*	*no*	*no*	*yes*
AV	*yes*	*yes/no**	*no*	*yes*

Table 2. A list of safety properties for visibility concerns related to shared memory accesses. (* – depends on a particular incarnation)

expected to make progress under lock semantics, and these programs must be guaranteed to make progress regardless of the underlying execution model. This is consistent with our correctness definition presented in Section 4.1 – two schedules generated for the same program under two different execution models cannot be equal if one of them is terminating and the other is non-terminating.

In languages like Java, different locks can be used to protect different accesses to the same shared data. In other words, no concurrency control is enforced if two accesses to the same location are protected by two different locks. As a result, two critical sections protected by different locks can execute concurrently, such as an outer critical section of thread `T1` and a critical section of thread `T2` in Figure 8(a).

Transactions, on the other hand, make no such distinction between critical sections. All transactions will appear serialized with respect to one another. Consequently, if the critical sections in Figure 8(a) were executed transactionally, the schedule presented in this figure could not have been generated. In a transactional implementation supporting pessimistic writers [3, 12], thread `T1` would acquire a write lock when writing `x` for the first time and release it only at the end of the outer critical section, making an intermediate read of `x` by thread `T2` impossible. One possible schedule that could be generated is presented in Figure 8(b). In accordance with isolation properties of transactions, propagation of both updates performed by thread `T1` is delayed until the end of the outermost critical section executed by thread `T1`. Observe that the schedule presented in Figure 8(b) would still be legal under lock-based execution if the runtime system used a different thread scheduling policy.

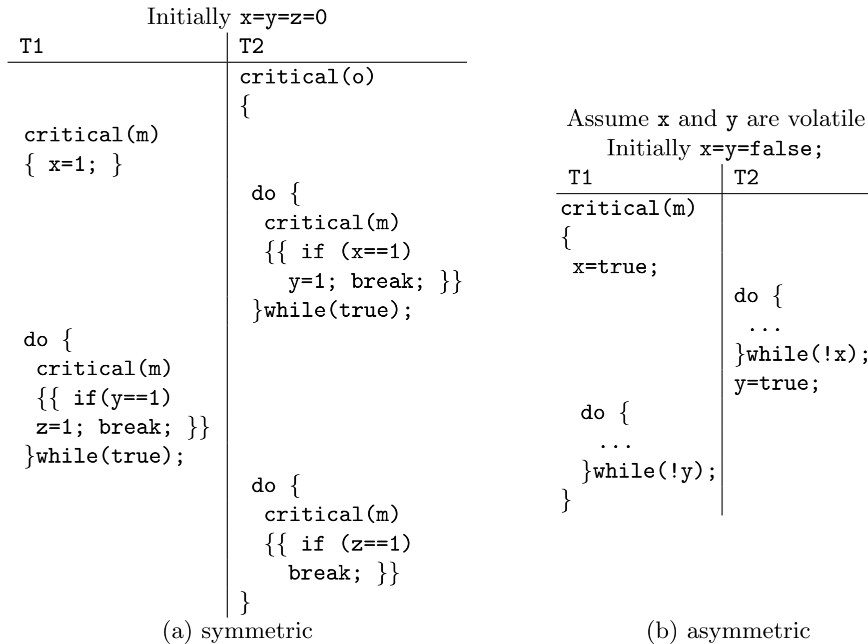
Thus, delaying propagation of updates can often be explained as a benign change in scheduler behavior. However, additional visibility-related safety properties must be defined to guarantee that lock semantics can be correctly supported by a transactional implementation in situations where this is not true. In the following code samples, an explicit "lock" parameter is used with the `critical` keyword in order to be able to express the difference between executions using transactions and executions using mutual exclusion locks. We assume that if transactions are used to execute critical sections then the "lock" parameter is ignored.

Visibility-related safety properties can be divided into two categories summarized in Table 2: symmetric dependent visibility (SV) and asymmetric dependent visibility (AV). Their definitions rely on the notion of control dependency be-

tween operations of different threads. We say that two operations are control dependent if, under all lock-based executions, the outcome of the first operation dictates whether the second one is executed.

Symmetric Dependent Visibility Dependent symmetric visibility (SV) concerns the case when all control dependent operations are executed inside of critical sections. The SV property is satisfied if a schedule can be generated in which, for every pair of control dependent operations, the second operation eventually sees the outcome of the first.

Consider the following example of a handshake protocol given in Figure 9(a). In this code the variables `x, y,` and `z` are used for communication. The two executing threads alternate waiting on variables to be set to one by spinning in while loops, making read and write operations on these variables control dependent. Under locks, the dependent symmetric visibility is obviously satisfied – the updates of `x, y,` and `z` will become visible to the respective threads allowing for the handshake to proceed. Consequently, the same visibility property must also be satisfied by P-SLE. However, if we executed this program using strong or weakly atomic transactions, the program would hang because transactions enforce isolation of shared accesses: execution of the outer transaction in thread T2 would prevent the update of `x` and `z` from being visible to the code it protects.



(a) symmetric (b) asymmetric

Fig. 9. Dependent visibility – is termination guaranteed?

Detecting potential visibility violations between critical sections is then tantamount to detecting control dependencies among data accesses executed within critical sections. We describe one solution to discovering symmetric dependent visibility violations in Section 6.

Asymmetric Dependent Visibility Unfortunately, examining only critical sections is insufficient since visibility issues can also arise between protected and unprotected code. Dependent asymmetric visibility (AV) addresses this case.

Consider the example given in Figure 9(b). Thread `T2`'s progress depends on making a result of thread `T1s` update of `x` available to thread `T2`. Since `x` is a volatile, lock semantics dictates that `Thread 2` must eventually see the result of thread `T1`'s update to `x`. Similarly, thread `T1` is guaranteed under lock semantics to see the result of thread `T2`'s update to `y`. Therefore, this program is guaranteed to successfully complete execution of both threads under lock semantics. At the same time, strong atomicity and certain incarnations of weak atomicity (such as those using write buffering) would prevent a write to `x` from being visible to thread `T2` until the end of the critical section in thread `T1`. If we run this program under such an implementation, the program will fail to terminate [4]. Once again we notice that the variables through which communication occurs are in fact control dependent. Discovering and remedying violations of the AV property is additionally complicated when compared to SV because of the to the asymmetry of control dependent operations (one of the operations is unprotected), as we describe in Section 6.

Our examples illustrate that neither strong nor weak atomicity satisfy the properties required from P-SLE to support lock semantics. As a result, existing transactional implementations must be modified to detect violations of lock semantics with respect to visibility concerns. We use a fallback mechanism to to remedy such problems whenever they are discovered.

5 Explicit Transactions

Based on the discussion presented in Section 4, we observe that Java's synchronized blocks can be supported by two implementations: non-transactional (mutual exclusion) and transactional (P-SLE). Now we have to consider the opposite – what are the implementations that can support user-defined atomic regions? Our task is much simpler here. We can choose the same transactional implementation, P-SLE, to support execution of both atomic regions and synchronized blocks. Since visibility-related properties of atomic regions are not as strictly defined as in case of Java's synchronized blocks, we only need to consider the properties of P-SLE that concern isolation and ordering, described in Section 4.2. The analysis of required properties for P-SLE defined in the last

[4] On the other hand, the execution under a weakly atomic model that exposes uncommitted values to other threads could lead to violation of the speculation safety property described in Section 4.2.

column of Table 1 reveals that this set corresponds to a so-called SGLA (Single Global Lock Atomicity) transactional semantics [9, 19]. SGLA is a middle-of-the-road semantics which is STM-implementation agnostic. SGLA provides a simple, more intuitive, semantics compared to weak atomicity, but does incur additional implementation-related constraints because it must provide stronger guarantees. By utilizing SGLA, our system is not tied to a particular underlying STM, as demonstrated in [19]. Atomic regions behave under SGLA as if they were protected by a single global lock, including treatment of entry and exit to every atomic region as a memory synchronization point for a unique lock with respect to visibility requirements defined by the JMM. This property allows us to trivially define a non-transactional implementation for atomic regions, *atomic serialization*, in which execution of atomic regions is serialized using a unique global mutual exclusion lock.

6 Implementation

Our implementation builds on an STM system using in-place updates, supporting optimistic readers and pessimistic writers, and providing strong atomicity guarantees [26]. Although our implementation leverages strong atomicity, any STM implementation that supports SGLA semantics is sufficient. We first briefly describe an implementation of the "base" transactional infrastructure, then discuss how multiple semantics can be supported within the same system, and finally describe modifications and extensions to the base transactional implementation needed to make it match properties required by P-SLE. Our implementation supports all of Java 1.4 including native method calls, class loading, reflection, dynamic thread creation, wait/notify, volatiles, etc. but does not currently support Java 1.5 extensions.

The base STM system extends the Java language into *Transactional Java* with an `atomic` primitive. Transactional Java is defined through a Polyglot [23] extension and implemented in pure Java. Our implementation utilizes the StarJIT compiler [2] to modify code of all critical sections so that they can be executed using either mutual exclusion locks or transactions. The run-time system, composed of the ORP JVM [7] and the transactional memory (TM) run-time library, provides transactional support and is tasked with handling interactions between transactions and Java monitors.

6.1 Base System

In the base STM system every access to a shared data item is "transactionalized" – mediated using a *transaction record*. In case of objects, a transaction record is embedded in the object header, and in case of static variables it is embedded in the header of an object representing the class that declares this static variable (access to all static variables of a given class is mediated through the same transaction record). The appropriate barriers implementing transactionalized

accesses are automatically generated by the JIT compiler and protect accesses to data objects performed inside of transactions.

Data objects can be either exclusively write-locked or unlocked. In case a given data item is unlocked, its transaction record contains a version number. In case a given data item is write-locked, its transaction record contains a *transaction descriptor pointer* pointing to a data structure containing information about the lock owner (*transaction descriptor*). These two cases can be distinguished by checking a designated low-order *version bit*. When a transaction record contains a transaction descriptor pointer, this bit is always unset since pointers are aligned. In case a transaction record contains a version number, this bit is always set because of the way version numbers are generated. All writes performed by a transaction are recorded in a transaction-local *write set* which allows a terminating (committing or aborting) transaction to release its write locks by incrementing version numbers for locations it has updated. All reads performed by a transaction are recorded in the transaction-local *read-set* which is used at transaction commit to verify validity of all the reads performed by the committing transaction – writes are automatically valid since they are preceded by acquisition of exclusive write locks. If this validation procedure is successful, the transaction is allowed to commit, otherwise it is aborted and re-executed.

Because the base STM system supports strong atomicity, appropriate barriers are generated for non-transactional data accesses. Non-transactional reads are allowed to proceed only when a given data item is unlocked. Non-transactional writes, in order to avoid conflicts with transactional accesses, behave as *micro-transactions*: they acquire a write lock, update the data item and release the write lock. Since the write is conceptually non-transactional, no explicit transaction owns the lock. Therefore, instead of a regular write lock, non-transactional writes acquire an *anonymous lock*. The anonymous lock is implemented by flipping a version bit to give the contents of the transaction record the appearance of a write lock [26].

We augment the base STM system to handle translation of both concurrency primitives (Section 6.2) and their interchangeability (Section 6.3). The STM was also extended with special types of barriers (Section 6.4) and a visibility violation detection scheme (Section 6.5). Our implementation spans all three parts of the STM system: ORP, StarJIT and the TM library.

6.2 Translating Concurrency Primitives

Both types of concurrency primitives (`synchronized` and `atomic`) are translated by the JIT compiler to use the same run-time API calls:

- `criticalstart(Object m, ...)`
- `criticalend(Object m, ...)`

Both API calls take an object as one of their parameters representing a Java monitor (i.e., a mutual exclusion lock) – either associated with the `synchronized` keyword or generated by the run-time system in case of `atomics`. The run-time

```
try {                                        while (true) {
 monitorenter(m)                              try {
 ...                                            criticalstart(m,..)
 monitorexit(m)                  ==>            ...
} catch (Throwable x) {                         if (criticalend(m,...)) continue;
 monitorexit(m)                               } catch (Throwable x) {
 throw(x)                                       if (criticalend(m,...)) continue;
}                                               throw(x)
                                              }
                                             break;
                                             }
```

Fig. 10. Translation of synchronized blocks

system, thus, has the ability to choose a specific implementation for critical sections – either transactional (by instructing the TM library to start or commit a transaction) or lock-based (by acquiring or releasing a monitor passed as a parameter). The remaining parameters are used to pass additional transactional meta-data.

A typical JVM (including ORP) handles synchronized blocks and synchronized methods differently. In case of synchronized blocks, the source-to-bytecode compiler (e.g., javac) generates explicit calls to the JVM's implementation of the bytecode-level `monitorenter` and `monitorexit` primitives. The synchronized methods, on the other hand, are simply marked as such in the bytecode and the JVM is responsible for insertion of the appropriate synchronization operations during the generation of method prologues and epilogues. In order to be able to treat synchronized methods and synchronized blocks uniformly, we modify the JIT compiler to wrap the body of every synchronized method in a synchronized block (using either `this` object or the "class" object as a monitor) instead of invoking synchronization operations in the method's prologue and epilogue.

Our translation of synchronized blocks and methods in the JIT compiler mirrors the code structure of atomic blocks generated at the source level by the Polyglot compiler [3]. Figure 10 depicts this translation as a pseudo-code transformation (with `monitorenter` and `monitorexit` operations represented as method calls) – the left-hand side shows code generated for acquisition of monitor `m` to support Java's synchronized blocks and the right-hand side the code after transformation. The actual translation involves an additional pass in the JIT compiler which transforms the control-flow graph – the pseudo-code description of the transformation is out of necessity somewhat simplified. The intuition behind this transformation is that `monitorenter(Object m)` and `monitorexit(Object m)` operations are translated to `criticalstart(Object m, ...)` and `criticalend(Object m, ...)` method calls. The existing bytecode structure is leveraged to handle exceptions thrown inside of a transaction. Re-executions of aborted transactions are supported through an explicit loop

inserted into the code stream – successful commit of a transaction is followed by a loop break.

During translation of original Java synchronization primitives, we also need to correctly disambiguate nesting. Our transactional API calls have one of two different versions selected depending on the execution context – outer (when no transaction is currently active) and inner (when executed in the context of an already active transaction). To discover the execution context, the JIT compiler performs a data flow analysis and builds a stack representation of monitor enter/exit calls for each node in the control flow graph. This allows us to identify the code fragments protected by transactions, and their nesting level. Additionally, every compiled method is marked as either transactional or non-transactional, depending on its calling context.

6.3 Coexistence of Transactions and Java Monitors

Ideally, Java monitors would be elided and all critical regions would be executed transactionally. Unfortunately, this is not possible for code that performs native method calls with side effects, triggers class loading and initialization, or contains critical regions that engage in inter-thread communication or spawn threads. In such cases, the fallback mechanism is utilized to use mutual exclusion rather than transactional execution.

The fallback operation is performed by aborting the currently running (outermost) transaction. The transaction is then restarted in *fallback mode* where concurrency is managed by acquiring and releasing Java monitors passed to transactional API calls as parameters. As a result, the implementation of the existing synchronization-related primitives, such as wait and notify operations, can be left largely untouched provided that they are only executed by *fallback transactions*.

Naturally, as a result of the fallback operation, some critical sections in the system are protected by Java monitors and others protected by transactions. In order to guarantee correct interaction between critical sections, every fallback transaction acquires transactional locks on all objects representing Java monitors it is using (to simulate transactional writes) and every regular transaction adds all its Java monitors to its read-set (simulating transactional reads). This prevents any regular transaction (translated from a Java monitor) from successfully completing if another thread is executing a critical section in fallback mode with the same monitor. The read-set of the regular transaction would become invalid, and the transaction itself would be aborted and re-executed. Notice that arbitrarily many regular transactions may execute concurrently, even if they share the same monitor.

6.4 Fallback Barriers

In the base STM system, the JIT compiler generates two versions of each method, transactional and non-transactional, containing appropriate versions of read and

T1	T2
`synchronized(n) {`	
`data=dummy;`	
	`critical(m) {`
	`tmp=data;`
`data=1;` // no barrier	...
`}`	
	`}`

Fig. 11. Incorrectly eliminated barrier thread (`T1` is in fallback mode, thread `T2` is transactional)

write barriers. The specific version of the method is then chosen at run-time depending on its invocation context. Read and write barriers occur on non transactional code since our implementation is strongly atomic. The JIT compiler eliminates unnecessary barriers where appropriate.

In our system, transactions in fallback mode behave differently than regular transactions - namely their execution must be faithful to Java monitor semantics. In order to reproduce Java monitor semantics, transactions in fallback mode have to ignore conflicts between each other while still properly interacting with regular transactions. Since concurrency control between a regular transaction and a transaction in fallback mode sharing the same monitor is mediated through this monitor, it does not need to be controlled at the level of data accesses. At the same time, according to the Java monitor semantics, no special concurrency control guarantees are provided between critical sections using different monitors. As a result, operations executed by a transaction in fallback mode should behave as if they were non-transactional, by blocking on reads of write-locked data items (to avoid speculation-related problems) and turning write operations into micro-transactions.

The read barriers for transactions executing in fallback mode can therefore be identical to non-transactional barriers. Unfortunately, turning write barriers into micro-transactions is surprisingly tricky. This is because code for fallback transactions has been compiled with regular transactional barriers. Optimizations may remove some barriers because they appear to be redundant. Consider the example shown in Figure 11 in which the second write of thread `T1` executing a transaction in fallback mode should cause an abort of the regular transaction executed by thread `T2`. Unfortunately, the transactional barrier at the second write is dominated by the barrier at the first write (in a regular transaction the first write would acquire a lock) and would be eliminated.

One option is to have the JIT compiler generate yet another version of all methods executed by transactions in fallback mode. However, this would lead to increase in code size, complicate the method dispatch procedure, and increase compilation time. Additionally, we would lose all benefits of barrier elimination optimizations. Therefore, we adopt a dynamic solution and re-use barrier code sequences for both regular transactions and transactions executing in fallback

mode. To do so, we introduce another version of the anonymous lock, the *fallback lock*. The fallback lock can be acquired only by a transaction executing in fallback mode and is held until this transaction is completed. If more than one fallback transaction wants to acquire a fallback lock for the same data item, the lock gets inflated - we need to count the number of writers and retain information about the version number. Regular transactions block when trying to access a data item locked by a fallback transaction. At the same time, non-transactional reads are allowed to access it freely, as are the non-transactional writes that additionally change fallback lock to an anonymous lock for the duration of the write operation [5]. A diagram illustrating transitions in a lock's state is given in Figure 12.

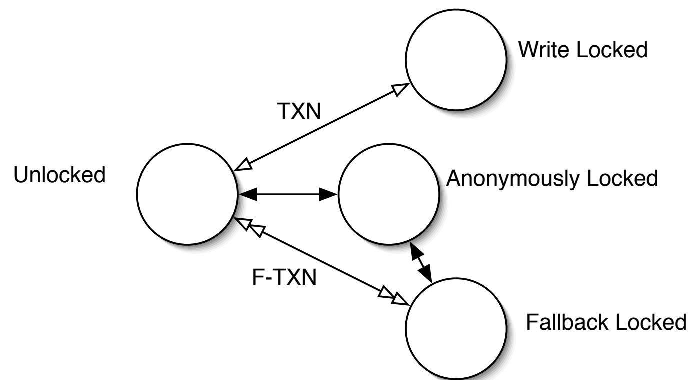


Fig. 12. Transitions made by non-transactional accesses are depicted by black arrows. White arrows represent transactional accesses – a single arrow by a bona fide transaction and a double arrow by a fallback transaction.

6.5 Detecting Dependent Visibility Violations

When dependent visibility violations occur, the observable effect is that execution becomes "stuck". We provide a mechanism to detect these situations rather than painstakingly tracking data and control dependencies between critical sections. Our specific solution differs slightly between cases of symmetric and asymmetric dependent visibility violations.

Symmetric Case In case some transaction T is (permanently) unable to complete its execution because it expects to see results computed by a different

[5] Fallback transaction must wait for anonymous lock to be released both when updating the value and when releasing the fallback locks

transaction T' (such as in the example presented in Section 4.3 in Figure 9(a)), the control dependencies between data access operations that belong to these transactions must form a cycle. Otherwise, if transaction T' was independent of transaction T and allowed to complete successfully, its computation results would be eventually made available to T removing the reason for it being stuck. This situation can be trivially generalized to the multi-transaction case. Obviously, we could employ full-fledged cycle detection to detect such situations, but because we assume these kinds of situation to be infrequent, we opt for a simpler solution and choose to utilize a time-out mechanism. If a transaction is unable to complete its data access operation after a pre-specified amount of time, it will be aborted and will re-execute in fallback mode. As a result of reverting to an implementation that is identical with Java monitors, the visibility violation cannot happen upon re-execution since Java monitor semantics prevents it automatically.

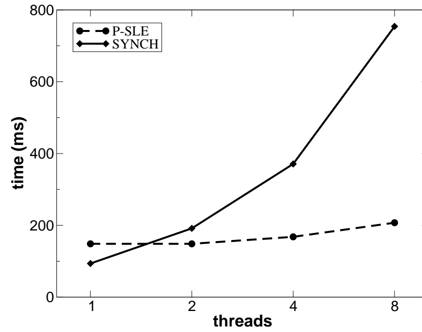
Asymmetric Case This case is a little more subtle, because execution can get stuck in a non-transactional code region (as illustrated in Section 4.3 in Figure 9(b)). Therefore, the run-time system has no transactional context available that could be aborted and re-executed in a different mode. The solution we adopt to handle this case is for the non-transactional data access operation that failed to successfully complete its execution after a pre-specified amount of time, and to request the transaction blocking this data access to abort and re-start in fallback mode. It is guaranteed that this request will be ultimately delivered since by definition there must be a control dependency that prevents the transaction from completing. The identity of the transaction blocking the data access is readily available from the transaction record that must contain its transaction descriptor in order for the access to be forbidden. Note that non-transactional execution can only get stuck on accesses to volatile variables – otherwise no happens-before edge forcing the visibility restriction would exist. Thus, we are obligated to modify only those data accesses that concern volatile variables.

7 Performance Evaluation

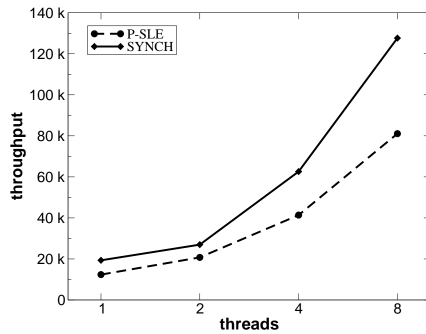
Our implementation is based on an STM system that has already been shown to deliver good performance for explicitly transactional applications [26]. In this section, we consider the performance characteristics of legacy applications executed with transactions and mutual exclusion.

In order to better understand the performance implications of our system, we chose three different benchmarks representing three different locking schemes:

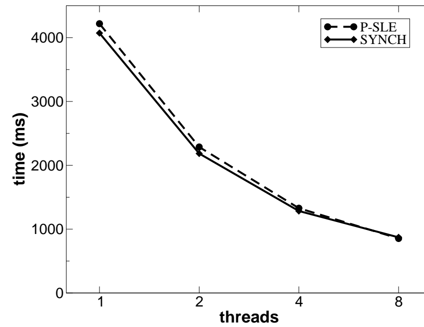
- OO7 [6]. This benchmark traverses and updates a shared tree-like data structure. Each each thread locks the entire shared data structure before performing a traversal (using a mixture of 80% lookups and 20% updates). This benchmark reflects a coding style that uses coarse-grain locks for concurrency control.



(a) OO7 execution time



(b) SPECjbb2000 throughput



(c) TSP execution time

Fig. 13. Preformance evaluation over multiple threads

- SpecJBB2000 [29]. This server application involves multiple threads operating over different objects (e.g., warehouses). Because operations of different threads are protected by different locks, this benchmark reflects a coding style that uses fine-grain locks for concurrency control.
- TSP. This implementation of traveling salesman involves threads that perform their searches independently, but share partially completed work and the best-answer-so-far via shared memory. It uses both fine- and coarse-grained concurrency control.

We ran all the benchmarks in two configurations: *P-SLE* which utilized our uniform transaction execution environment, transparently translating all synchronized blocks into atomic regions and dynamically falling back where necessary and *Synch* which used mutual exclusion provided by original implementation of Java monitors. Synch represents the original benchmark without the additional overhead of transactional instrumentation and fallback. All our experiments have been performed on an Intel Clovertown system with two 2.66 GHz quad-core processors for a total of eight hardware threads and 3.25 GB of RAM running Microsoft Windows XP Professional with Service Pack 2.

The performance of our system has met our expectations. When running the lock-based version of OO7 benchmark transactionally under the P-SLE configuration, the system was able to automatically extract additional parallelism and significantly improve performance over executions using coarse-grain mutual exclusion locks (Figure 13(a)). Since our implementation is based on a strongly atomic implementation, there is a certain amount of overhead that is expected [26] when only small amount of additional parallelism is available compared to executions using mutual-exclusion locks. However, note that even though the absolute performance of SPECjbb2000 when executed transactionally does not quite match its performance when executed using mutual-exclusion locks, the scalability characteristics in both cases is virtually the same (Figure 13(b)). Finally, the performance of the TSP benchmark is almost identical, regardless of whether it is executed transactionally or using mutual-exclusion locks (Figure 13(c)).

Given that our system is based on a strongly atomic engine, our performance evaluation results reflect very similar trends to those reported in [26]. In our prior case study benchmarks were modified by hand to use explicit atomic blocks under a strong atomicity model. Using our uniform transactional execution environment we were able to avoid by hand translation and ensure safety while improving both scalability and performance.

8 Related Work

Recently there have been many proposals for Software Transactional Memory [5, 11, 13, 14, 17, 18, 20, 22, 24, 26]. Such systems, unfortunately, provide limited or no support to compose transactions with locks. The Haskel STM [11] utilizes the type system to prevent any I/O actions within a transaction. Although we suspect such a restriction would allow for the *safe* composition of locks and atomic regions, it defacto limits the use of libraries that perform I/O in transactions. Other systems do not explicitly prevent composability of concurrency constructs, but they leave their interactions undefined. Interactions can vary based on the transactional implementation. Weakly atomic STMs, such as [13], suffer from subtle visibility and isolation anomalies [4], where as strongly atomic systems prevent the use wait/notify primitives. As such, program semantics vary based on the virtual machine's implementation of Java monitors as well as the guarantees provided by the STM itself. Programs must be hand-tuned for each system and virtual machine pairing.

Previous work which attempts to combine Java monitors and transactions [30, 31] place restrictions on programmers. Notably, programs must be race-free, even if races are benign. Such restrictions prevent the use of programming paradigms such as privatization. In such systems, programmers are forced to examine all interactions between transactions and locks to guarantee safety.

Other approaches [27] attempt to mirror lock based semantics by providing programmers with additional primitives to break transactional properties. Potential interactions between threads are tracked through the type system and

reported to the programmer. The programmer is required to establish consistency at specific points within the transaction. Unfortunately, the programmer must reason about the *transitive* effects of a given transactional region. Transactional regions maybe called from many different contexts and it is unclear if a break of isolation in one context is compatible with another.

9 Conclusions

We have presented the design and implementation of the first uniform transactional execution environment for Java programs. We have explored implications of executing *arbitrary* lock-based Java programs transactionally. We have also presented techniques that allow explicit transactional constructs, such as atomic blocks, can be seamlessly integrated into an existing programming language. We have presented performance evaluation of our system that demonstrates its ability to extract additional parallelism from lock-based applications by executing them transactionally, providing better performance when coarse-grain locks are used and providing performance approaching those of mutual-exclusion locks in case of fine-grain locking.

References

1. M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL 2008*.
2. A.-R. Adl-Tabatabai, J. Bharadwaj, D.-Y. Chen, A. Ghuloum, V. S. Menon, B. R. Murphy, M. Serrano, and T. Shpeisman. The StarJIT compiler: a dynamic compiler for managed runtime environments. *Intel Technology Journal*, 7(1), 2003.
3. A.-R. Adl-Tabatabai, B. T. Lewis, V. S. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI 2006*.
4. C. Blundell, E. C. Lewis, and M. Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), Nov. 2006.
5. C. Blundell, E. C. Lewis, and M. Martin. Unrestricted transactional memory: Supporting i/o and system calls within transactions. Technical Report CIS-06-09, University of Pennsylvania, Department of Comp. and Info. Science, 2006.
6. M. J. Carey, D. J. DeWitt, C. Kant, and J. F. Naughton. A status report on the OO7 OODBMS benchmarking effort. In *OOPSLA 1994*.
7. M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth. Open Runtime Platform: A Flexible High-Performance Managed Runtime Environment. *Intel Technology Journal*, 7(1), February 2003.
8. J. Gosling, B. Joy, G. Steele, Jr., and G. Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000.
9. D. Grossman, J. Manson, and W. Pugh. What do high-level memory models mean for transactions? In *MSPC 2006*.
10. T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA 2003*.
11. T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *PPoPP 2005*.

12. T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI 2006*.
13. M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC 2003*.
14. M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA 2006*.
15. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.
16. J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL 2005*.
17. V. J. Marathe, W. N. Scherer, and M. L. Scott. Adaptive software transactional memory. In *International Symposium on Distributed Computing 2005*.
18. V. J. Marathe, W. N. Scherer, and M. L. Scott. Design tradeoffs in modern software transactional memory systems. In *LCR 2004: Languages, Compilers, and Run-time Support for Scalable Systems*.
19. V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Single global lock semantics in a weakly atomic stm. In *TRANSACT 2008*.
20. M. Moir. Hybrid hardware/software transactional memory. http:// www.cs.wisc.edu/ rajwar/tm-workshop/TALKS/moir.pdf, 2005.
21. J. E. B. Moss and A. L. Hosking. Nested transactional memory: model and preliminary architecture sketches. In *SCOOL 2005*.
22. Y. Ni, V. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open Nesting in Software Transactional Memory. In *PPoPP 2007*.
23. N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: an extensible compiler framework for Java. In *CC 2005: International Conference on Compiler Construction*.
24. M. F. Ringenburg and D. Grossman. AtomCaml: first-class atomicity via rollback. In *ICFP 2005*.
25. B. Saha, A.-R. Adl-Tabatabai, R. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *PPoPP 2006*.
26. T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and S. Bratin. Enforcing isolation and ordering in stm.
27. Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with Isolation and Cooperation. In *OOPSLA 2007*, pages 191–210.
28. M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. Technical Report 915, University of Rochester, Computer Science Dept., 2007.
29. Standard Performance Evaluation Corporation. SPEC JBB2000, 2000. See `http://www.spec.org/jbb2000`.
30. A. Welc, A. L. Hosking, and S. Jagannathan. Transparently reconciling transactions with locking for java synchronization. In *ECOOP 2006*.
31. A. Welc, S. Jagannathan, and A. L. Hosking. Transactional monitors for concurrent objects. In *ECOOP 2004*.