A Composable Deadlock-Free Approach to Object-Based Isolation

Shams Imam(✉), Jisheng Zhao, and Vivek Sarkar

Department of Computer Science, Rice University, Houston, USA
{shams,jisheng.zhao,vsarkar}@rice.edu

Abstract. A widely used principle in the design of concurrent programs is isolation – the property that a task can operate on shared data without interference from other tasks. In this paper, we introduce a new approach to object-based isolation that is guaranteed to be deadlock-free, while still retaining the rollback benefits of transactions. Further, our approach differentiates between read and write accesses in its concurrency control mechanisms. Finally, since the generality of our approach precludes the use of static ordering for deadlock avoidance, our runtime ensures deadlock-freedom by detecting and resolving deadlocks at runtime automatically, without involving the programmer.

Keywords: Object-Based isolation · Deadlock freedom · Lock composition · Read-Write locks · Delimited continuations

1 Introduction

Designing and implementing correct and efficient concurrent programs is a notoriously challenging task due to the possibility of data races. Programs must use concurrency control mechanisms to ensure that multiple threads of execution do not interfere with each other while sharing data in memory. One approach for enforcing mutual exclusion is to use critical sections that execute in isolation with respect to other *interfering* critical sections. Isolation is the property that a thread can access shared data without interference from other threads.

Threads use locks to guard the operations performed while the lock is held; this enforces isolation properties of a thread's guarded operations. The dominant concurrency control mechanism in high-level languages, such as Java and C#, are mutual-exclusion locks [2]. Parallel programming models (e.g. OpenMP 4.0 [20], Cilk [8]) also rely on locks for implementing mutual exclusion. In fact, there is comprehensive empirical evidence that programmers almost always use mutual-exclusion locks to enforce isolation properties [7]. Transactional memory offers a promising alternative to lock-based synchronization as a mechanism for isolation. A programmer can reason about the correctness of code within a transaction and need not worry about interactions with other concurrently executing transactions [14]. However, re-execution of conflicting transactions, and the logging of data accesses to prepare for the possibility of rollback, add overhead and often lead to poor performance even in the presence of moderate contention.

J.L. Träff et al. (Eds.): Euro-Par 2015, LNCS 9233, pp. 426–437, 2015.
DOI: 10.1007/978-3-662-48096-0_33

The focus of this work is to provide a deadlock-free construct to support shared-exclusive object-based isolation in concurrent programs. When threads can not coordinate their accesses to shared data, deadlocks can occur while acquiring isolation privileges. When a deadlock can occur, a dynamically assigned low-priority thread is forced to roll back and release privilege(s) it is holding that is preventing a high-priority thread from making progress. Once released, this allows the high-priority thread to acquire the privilege and make progress. After the high-priority thread releases the conflicting privileges, the low-priority thread can resume execution. Our construct combines the features of transactions and shared-exclusive locks to resolve deadlocks and to minimize re-executions due to conflicts. It also enables promotion of shared privileges to exclusive privileges by rolling back part of the computation and re-executing it with an exclusive privilege.

In summary, the contributions of this paper are as follows:

- We introduce object-based isolation as a high-level construct for deadlock-free shared-exclusive mutual-exclusion.
- We describe an implementation approach for object-based isolation that resolves deadlocks at runtime, exploits the rollback benefits of transactions, and differentiates between read and write accesses in its concurrency control mechanisms.
- We compare the performance obtained by our implementation of object-based isolation with that of Java's `synchronized` statement, the regular and shared-exclusive locks available in the JDK [13], and with the Multiverse STM library [18].

2 Background and Motivating Example

Large multi-threaded programs that involve concurrency control via the use of multiple locks can be challenging to write. Deadlocks can occur when multiple threads need the same locks but obtain them in a different order. Always acquiring locks in a consistent order ensures that programs will not deadlock. But this can be challenging (or even impossible) to ensure as the dynamic dispatch capabilities or library composition features in many languages make it difficult to know a program's exact call graph structure at compile-time.

Using read-write (shared-exclusive) locks can significantly improve parallel performance if the protected data is read frequently and modified only occasionally. They can be acquired either for reading or for writing: multiple readers may hold the lock simultaneously, but writers must acquire exclusive ownership of the lock. Along with deadlocks, an issue while composing software components is the need to promote read privileges to write privileges and vice versa. Promoting (demoting) a write privilege to a read privilege carries no restrictions and can be supported trivially with reentrant behavior. However, promoting a read privilege to a write privilege is usually not permitted as doing so can lead to inconsistent behavior and is prone to deadlocks. As we will see in Sect. 3.3, our construct also supports promotion of read privileges to write privileges.

2.1 Motivating Example

Our motivating example is the classic bank transaction; a transaction must debit one account and credit another with a particular amount of money when legal to do so. For proper accounting, it is essential that either both operations succeed or neither operation succeeds. This means that both operations should be performed with transactional semantics to ensure the integrity of the system's state.

Listing 1.1. Classic bank transaction example using read-write locks.

```
1  class BankTransactionRWLock {
2    def trySafe(from, to, amount) {
3      lock (from.readLock) {
4        if from.balance() > amount
5          transfer(from, to, amount)
6          return true
7        else
8          return false
9    } }
10   def transfer(from, to, amount) {
11     val low = min(from, to)
12     val high = max(from, to)
13     lock (low.writeLock) {
14       lock (high.writeLock) {
15         from.debit(amount)
16         to.credit(amount)
17   } } } }
```

Listing 1.2. Classic bank transaction example using transaction memory solution with `atomic` blocks.

```
1  class BankTransactionAtomic {
2    def trySafe(from, to, amount) {
3      atomic {
4        if from.balance() > amount
5          transfer(from, to, amount)
6          return true
7        else
8          return false
9    } }
10   def transfer(from, to, amount) {
11     atomic {
12       from.debit(amount)
13       to.credit(amount)
14   } }
15 }
```

Listing 1.1 uses read-write locks to ensure fine-grained synchronization to increase concurrency. To avoid deadlocks, it uses ordering of the bank accounts to retrieve the locks (lines 11–12). However, composing of `trySafe` and `transfer` can still lead to deadlock since the locks individual instances can still be acquired out of order from multiple calls. In addition, deadlocks will also occur in systems that do not allow promotion of read privileges (in `trySafe()`) to write privileges (in `transfer()`).

Listing 1.2 displays the same example written using `atomic` blocks that offer software transactional memory (STM) support. Deadlock is not possible in this example as transactions never wait for one another; at least one transaction is guaranteed to succeed in the presence of conflicts. In STMs, a conflict occurs when two concurrent uncommitted transactions perform conflicting read or write operations on the same bank accounts. However, the likelihood of aborting due to intervening conflicting commits increases in longer running transactions or in write-heavy workloads, causing deterioration in performance.

3 Object-Based Isolation as a High-Level Construct for Concurrent Programming

In this section, we introduce Object-Based Isolation (OBI) as a high-level construct for concurrent programming. Our goal for OBI is to combine the programmability of scoped `synchronized` blocks with the efficiency of read-write locks

(RWLs) and the semantic guarantees of transactional execution (i.e. isolation, deadlock freedom, optimistic concurrency). As with locks, mutual exclusion is only guaranteed between instances of `isolated` statements; no such guarantees exist between `isolated` and non-`isolated` statements. Assuming that there are no data races between `isolated` and `isolated`/non-`isolated` statements, an `isolated` statement executing in parallel is guaranteed to produce the same answer (for the same input state) as when no other task/thread is executing at the same time [14]. As with transactions (but not with locks), the programmer is spared the burden of guaranteeing deadlock freedom – that burden is passed on to the implementation instead.

As we will see, our proposed `isolated` statement is scoped, and (unlike transactions) allows the user to specify a list of objects with read or write (R/W) modes for which isolation is desired. Two `isolated` statements are only guaranteed to execute in mutual-execution if they have a non-conflicting intersection in their shared-exclusive object sets. This allows `isolated` statements to execute critical sections that are guarded by explicitly specified objects, unlike in transactions where critical sections appear to be guarded globally leading to deadlock scenarios [16]. In transactional memory, a data access pattern with frequent writes to shared data will induce numerous aborts; such issues do not arise with `isolated` statements. `isolated` statements can be nested, and inner statements can add to the set of objects acquired. No total order is imposed on the nested `isolated` object list. This capability allows for the expression of "non-cautious" concurrency patterns[1] [21].

3.1 `isolated` Statements

Listing 1.3. Classic bank transaction example using `isolated` blocks with read and write privileges.

```
1  class BankTransactionIsolated {
2    def trySafe(from, to, amount) {
3      isolated(read(from)) {
4        if from.balance() > amount
5          transfer(from, to, amount)
6          return true
7        else
8          return false
9    } }
10   def transfer(from, to, amount) {
11     isolated(write(from), write(to)) {
12         from.debit(amount)
13         to.credit(amount)
14   } } }
```

The motivation for OBI is that there are many cases when the programmer knows the shared or exclusive mode for the set of objects that will be accessed in the body of an `isolated` statement. The specification of these modes in the `isolated` argument object set helps the runtime by explicitly stating the objects that need to be tracked. Listing 1.3 displays the bank transaction example from Sect. 2.1 using `isolated` statements. As with Java's `synchronized` construct, `isolated` is reentrant and scoped guaranteeing the absence of dangling unlock operations. Like RWLs, `isolated` statements can acquire (R/W) access privileges on the argument

[1] Cautious patterns require all reads to shared data to performed before mutations to any of them.

object. Unlike Java's RWLs, object sets in nested `isolated` blocks allow promotion of an object's access privilege from shared mode to exclusive mode (Sect. 3.3). For nested `isolated` constructs we follow open nested semantics [9], they do not enforce atomicity. Our prevention scheme (Sect. 3.4) avoids deadlock while acquiring privileges. While sequential composition of transactions to form a single, larger transaction can cause deadlocks [16], composition of `isolated` statements can never cause a deadlock.

3.2 Execution Mechanism

Since we allow free composition of `isolated` statements, we cannot guarantee an order in the acquisition of privileges by `isolated` statements. Similarly, it is impossible to prevent scenarios where a read privilege needs to be promoted to a write privilege. As mentioned in Sect. 2.1, both these behaviors are prone to deadlocks. As a result, our approach dynamically detects and resolves deadlocks by allowing instances of `isolated` statements to *abort* by rolling back and re-executing with possibly modified privileges when it is safer to do so.

Listing 1.4. Simple `Counter` that supports the *clone-merge* protocol. For simple data structures, these can be automated by a compiler.

```
1 class Counter(var value) {
2   def increment(amount) {
3     value += amount
4   }
5   def clone() {
6     return new Counter(value)
7   }
8   def merge(other: Counter) {
9     this.count = other.count;
10 } }
```

Roll backs during the execution of a thread can leave shared data in an inconsistent state. Two key observations help resolve this concern. Firstly, the inconsistency occurs only due to objects that were being modified, i.e. those objects executing with a write privilege. We resolve this issue by employing a *clone-merge* protocol where a clone of the object is created and used inside the body of the `isolated`. When the `isolated` statement completes successfully, this private clone is trivially merged back into the source object since only one `isolated` gets to run with write privileges. If an `isolated` statement aborts, the clone is not merged and discarded. Listing 1.4 displays a simple integer counter class that supports cloning and merging. Relying on the clone-merge protocol limits the applicability of our approach on classes for which the source code is not available and a clone method is cannot be generated automatically. The other approaches, such as transactional memory and locks, do not suffer from this limitation.

Secondly, we need a scheme to dynamically identify the target program points when a computation is rolled back. To address this concern, we use delimited continuations (DeCont) [6] to roll back the computation. Each `isolated` statement executes as a DeCont and the call stack is recursively unwound from nested `isolated` statements to a target `isolated` statement during rollback. Since our approach works on clones and only commits the results when successful, we can handle roll backs very easily. When re-executing an `isolated` statement, a new DeCont is created and executed.

With OBI, tracking every read and write is obviated as the programmer explicitly declares the read and write object sets. Unlike transactions where

every mutated object is committed at the end of the transaction, only object sets opened in write mode are committed at the end of an `isolated` statement. We employ a pessimistic control policy [9], where `isolated` statements only execute their statements once they are guaranteed it is safe to do so. While the likelihood of intervening conflicting commits increases in longer running transactions, such situations do not arise with `isolated` statements. The guarantee relies on the use of read-write lock semantics where an `isolated` statement is forced to wait until it successfully acquires the desired read-write privilege.

In traditional transactions, a conflict is resolved by aborting and re-executing or delaying one of the conflicting transactions. Similarly in conflicting scenarios, at least one `isolated` statement aborts and re-executes later. However, the difference with transactions lies in the situations identified as conflicts. With our OBI, there are two scenarios that can cause conflicts. The first is when we dynamically detect that a nested `isolated` statement is attempting to acquire a write privilege for a previously acquired read privilege. The second is when `isolated` statements participate in a deadlock cycle while attempting to obtain a privilege as is possible in the bank transaction example.

3.3 Read-to-Write Promotion

It is not safe to simply promote a read privilege to a write privilege. Even if the write privileges are promoted serially without rollback, the invariants that were true while executing with read privileges may no longer hold due to intervening writes. To address this issue, we recursively roll back the computation to the outermost `isolated` statement that acquired the read privilege on the object. The privilege for that instance of the `isolated` statement is dynamically updated to a write privilege, and the statement re-executed. Listings 1.5 and 1.6 display an example of the transformation that happens dynamically at runtime before and after the read-to-write promotion, respectively. The read privilege for `x` in the outermost `isolated` on line 2 is promoted to a write, and the statements in lines 3 to 7 re-executed.

Listing 1.5. Snippet with nested `isolated` statements that require read-to-write promotion. Note that these promotions could occur across deeply nested blocks within different function calls.

```
1 isolated (write(w)) {
2   isolated (read(x)) {
3     isolated (read(y)) {
4       isolated (read(x)) {
5         isolated (write(x)) {
6           ...
7 } } } } }
```

Listing 1.6. Snippet with nested `isolated` statements after read-to-write promotion. Note that this promotion happens to the dynamic instance being executed and not to the static version of the code.

```
1 isolated (write(w)) {
2   isolated (write(x)) { //promoted
3     isolated (read(y)) {
4       isolated (read(x)) { //
            unchanged
5         isolated (write(x)) {
6           ...
7 } } } } }
```

3.4 Deadlock Resolution

While executing nested `isolated` statements, we rely on dynamically detecting and resolving deadlocks. We associate unique ids with threads to prioritize them; these priorities are used to resolve conflicts. When a thread acquires a read or write privilege on an object, the thread is registered as an owner of the privilege on the object. When another thread attempts to acquire the same privilege and fails, it compares its priority with the owner of the lock. If it has a lower priority, it aborts by rolling back its computation releasing any read or write privileges it had acquired. If the failed thread has a higher priority order, it re-attempts to acquire the lock. The lower priority thread will eventually release the lock, either by aborting as mentioned above (Sect. 3.2) or by completing successfully, and allow the higher priority thread to continue with its execution. This prioritization strategy introduces unfairness in its scheduling policy but allows livelocks to be ruled out in our `isolated` construct.

4 Implementation

In this section, we discuss our implementation of OBI presented in Sect. 3. Despite any productivity promises, an abstraction must be implementable in an efficient and scalable fashion for it to be accepted by programmers. The `isolated` construct must incur a sufficiently low overhead to be useful in practice, especially for small transactions. Our implementation [12] is a Java-based task-parallel runtime that supports `async-finish` style computations, though our ideas can also be implemented in other thread-based languages including C/C++. Our implementation conforms to the constraints imposed by a standard Java Virtual Machine (JVM). In particular, standard JVMs do not provide support for DeConts or for storing and restoring the stack. The DeConts created are thread independent and can be resumed on any worker thread.

We use an extended version of the open source bytecode weaver provided by the Kilim framework [23] to support DeConts. The Kilim bytecode weaver works by transforming the code of methods which can trigger rollback. It recognizes such methods by the presence of a `SuspendableException` exception in the method signature. It is important to note that no actual exceptions are thrown or caught which minimize the overhead of capturing and resuming continuations. Instead, the transformation performed is similar to a continuation passing style transformation, except that only methods that can suspend are transformed.

The runtime maintains a pool of custom RWLs, this pool can be extended to be one per user object. However, for ease of implementation we maintain a fixed size list and hash objects to one of the locks. When an `isolated` block requires a read or write privilege on an object, it hashes the object to a read-write lock and attempts to acquire the read or write privilege. During a deadlock or a read-to-write promotion conflict, the computation is rolled back by capturing the continuation. In other failed attempts, the task suspends and registers itself on a wait list and is resumed by moving itself into the work queue when the

lock is available. Using continuations allows the worker thread to execute other ready tasks while suspended tasks are stored away in a separate queue.

5 Experimental Results

In this section, we present an experimental evaluation of the `isolated` construct introduced in this paper. We compare it against existing mutual-exclusion constructs available in the JDK - `synchronized` statement, JDK's ReentrantLock, and JDK's ReentrantReadWriteLock [13]. The JDK variants of the benchmarks were written to ensure there is no conflict (deadlock or read–to-write promotion) scenario during execution. We also compare our implementation on micro-benchmarks against the Multiverse library [18] whose STM implementation is based on [3]. We ran the benchmarks on four eight-core IBM POWER7 processors running at 3.8 GHz each. Each node contains 256 GB of RAM; the software stack includes IBM Java SDK Version 7. The JVM configuration flags used were (`-XX:-UseGCOverheadLimit -Xmx16384m -XX:+UseParallelGC -XX:+UseParallelOldGC`). Each benchmark was configured to run using 32 worker threads and run for thirty iterations in six separate JVM invocations. The arithmetic mean of the best fifty execution times (from the hundred and eighty iterations) are reported. Using the best execution time allows us to minimize the effects of JVM warm up, just-in-time compilation, and garbage collection.

5.1 Micro-Benchmarks

First, we compare the performance of the `isolated` construct on four microbenchmarks. The first microbenchmark uses Bank Transaction (BT) like those shown in Listings 1.1, 1.2, and 1.3. The second is an integer counter (CTR) microbenchmark where the increments to the counter are protected in mutual-exclusion blocks. The last two microbenchmarks are a concurrent read-write benchmarks on dictionary (CD) and sorted linked list (CSLL) data structures where the write percent is kept at 10 percent. The read and write operations in the CD benchmark takes $O(1)$ time while in the CSLL benchmark they take $O(N)$ time. All four lock variants perform similarly in BT, CTR, and CD as the critical section blocks are relatively short. The Multiverse STM version performs poorly compared to the other variants in the nested transactions BT benchmark. In CSLL, the critical section blocks take $O(N)$ time, hence the read-write lock version performs better than the reentrant lock version. The `synchronized` version performs better than the lock versions. The `isolated` version performs better as its use of continuations avoids blocking the worker threads allows all available read requests to be processed when there are no pending writes. The performance benefit comes from avoiding the need to context switch threads. Multiverse STM performs best on CSLL with a single transaction encapsulating the entire read or write operation (Fig. 1).

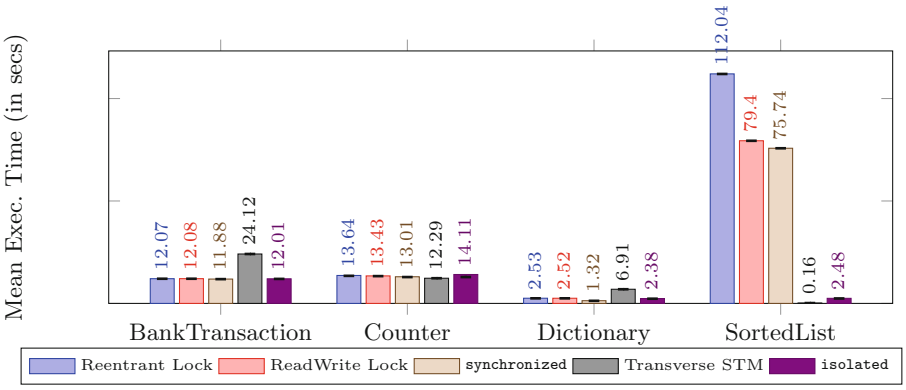

Fig. 1. Bank Transaction involves 6 million transactions on 8 thousand bank accounts. Counter includes 6 million increment operations each on 8 counter objects. The Dictionary benchmark 1 million operations with a write percent of 20 (split equal in `put` and `remove`) and remaining as read `get` operations. The SortedList benchmark 100 thousand operations with a write percent of 20 (split equal in `add` and `remove`) and remaining as read `get` operations. The y-axis represents program execution time, hence, smaller is better.

5.2 Macro-Benchmarks

We consider two larger benchmarks: Labyrinth and Parallel Breadth-First Search (BFS). The Labyrinth benchmark from the STAMP suite [17] is characterized by long transaction lengths, large read-sets, large write-sets, long transaction times,

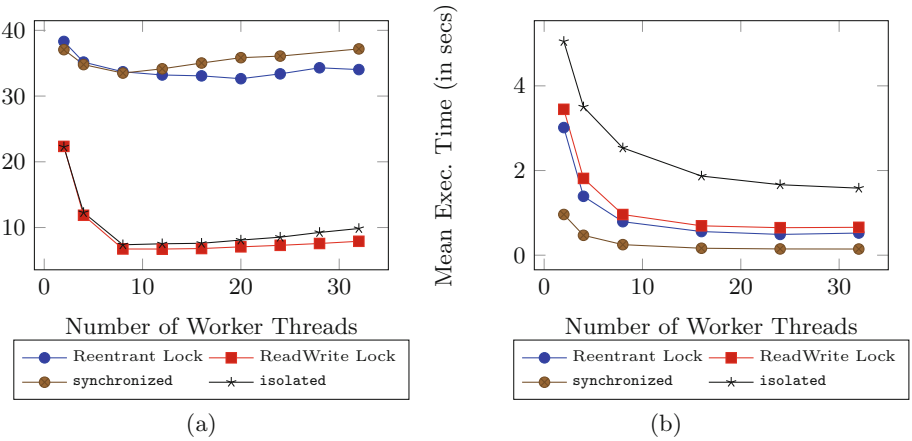

Fig. 2. Labyrinth (left) using the configuration 512 randomly generated inputs on dimension of $512 \times 512 \times 7$. Simple BFS (right) on a randomly generated connected graph with 500 thousand nodes and 5 million edges. The y-axis represents program execution time, hence, smaller is better.

and very high contention. The high contention causes the reentrant lock and `synchronized` versions to perform poorly with very low scalability. The read-write lock and `isolated` versions show improved performance as they allow multiple read requests to proceed in parallel. Simple BFS is a naive parallel implementation of the sequential BFS algorithm. In the BFS benchmark, the read-write lock, reentrant lock, and `synchronized` variants allocated one lock per graph node. The `isolated` version in our implementation shows higher overheads as it relies on the runtime to allocate a handful of locks (256 to be exact) and hashes on them (Fig. 2).

Note that the Java VM provides native support for `synchronized` statements and locks, but not for continuations. Our implementation of `isolated` uses DeConts without modifying the VM; the performance of our implementation would be greatly improved by using native support for DeConts in the VM. Work by Stadler et al. [24] to provide such native support in a Java VM reported over two orders magnitude speedup on micro-benchmarks compared to a bytecode transformation approach.

6 Related Work

Most of the state-of-art lock-free language constructs are based on transactional memory (TM) systems [10]. Both hardware transactional memory (HTM) [10] and software transactional memory (STM) [9] guarantee lock-free and deadlock avoidance by employing a rollback. By using TM, users can employ both coarse-grain and fine-grain parallelism, but have to pay for the overhead of rollback, especially for contention intensive (i.e. high conflict rate) critical sections. Recently, Aida [15] provides a high-level minimalistic programming model similar to Transactional Memory [10], with a single construct (`async isolated`) to define blocks of code to be executed concurrently and in isolation. Aida guarantees deadlock-freedom and livelock-freedom. Both STM and Aida need compiler support to instrument the memory accesses and enable the rollback mechanism.

Galois [19] is a runtime library-based approach, it provides library constructs called optimistic iterators for packaging optimistic parallelism as iterations over sets and for specifying the scheduling policy, and uses runtime scheme for detecting the conflicting shared data accesses and recovering from those unsafe access (i.e. rollback). Rajwar and Goodman based their technique on the observation that programmers often used coarse-grained locking to be sure "all bases are covered" and that programs can often run correctly even if the lock is never acquired [22]. Hence, the conservative locking strategies that programmers often use to ensure the correctness can frequently be elided dynamically, provided that one can detect and roll back concurrent updates that would have been prevented had the locking been performed. They built this work on speculative lock elision by automatically wrapping transactions around the critical sections of sequences of instructions detected at runtime as locks.

Lock inference [1,11] is a compiler-assisted approach to building efficient critical sections while also ensuring correctness. The basic idea is to employ compile-time analysis to identify the "really necessary" locks for the given critical section.

The efficiency depends on whether the static program analysis can precisely identify the lock set that should be applied to the critical section, i.e. in the presence of ambiguous object references a coarse-grain locking has to be chosen.

In this paper, we introduced the object-based isolation that is a runtime based mechanism provided to the user to efficiently build parallel application with guaranteed deadlock avoidance and livelock-freedom. The user interface is a language construct-like API, the advantage of this approach is that it provides the user a simple interface to build fine-grain locking based parallel applications, i.e. users can explicitly specifiy the mutual-excluded objects via our APIs. This is also a lightweight isolation support compared with Aida [15] and STM [10] which backups all objects within the language constructs specified scope. The user does not need to specify task scheduling strategies like Galois, our parallel runtime implicitly supports efficient scheduling mechanism (i.e. work-stealing).

Other approaches have exploited using lock-based implementation to improve the efficiency of STM. Ennals utilizes a hybrid policy where a pessimistic approach is used for write privileges, whereas an optimistic approach is used for read accesses [5]. Dice and Shavit used an optimistic control policy and only obtain locks before committing their writes, aborting the transaction if necessary [4]. Object-based isolation employs a pessimistic control policy for both read and write privileges (i.e. obtain the read or write privileges eagerly). In [5], deadlocks are detected while acquiring locks and a transaction can request another transaction to abort. [4] employs timeouts (for acquiring process) to abort a transaction and avoid deadlocks. Our approach does not require signaling the other task, a lower priority task cooperatively aborts its transaction to allow another task to make progress. The rollbacks of `isolated` statements happen only till the relevant outer boundary (for example, as shown in the example in Listing 1.6), unlike the above mentioned transaction approaches where the outermost transaction needs to be aborted.

7 Summary

We introduced a new composable approach to object-based isolation that is guaranteed to be deadlock-free, while still retaining the rollback benefits of transactions. Further, our approach differentiates between read and write accesses in its concurrency control mechanisms. Our construct incurs a cost for creating and merging clones which may, for some (large) data structures, require effort to implement efficiently by the programmer. We are currently exploring the possibility of implementing the `isolated` construct with native VM support to extract more performance. We, ambitiously, envision a scenario where the `synchronized` statement is replaced by the `isolated` construct semantics in modern programming languages.

Acknowledgments. We are very grateful to the anonymous reviewers, John Mellor-Crummey, Karthik Murthy, and Rishi Surendran for their insightful comments on early drafts that substantially improved the paper.

References

1. Cherem, S., Chilimbi, T.M., Gulwani, S.: Inferring locks for atomic sections. In: PLDI 2008, pp. 304–315 (2008)
2. Demsky, B., Lam, P.: Views: synthesizing fine-grained concurrency control. ACM Trans. Softw. Eng. Methodol. **22**(1), 4:1–4:33 (2013)
3. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
4. Dice, D., Shavit, N.: Understanding tradeoffs in software transactional memory. In: CGO 2007, pp. 21–33. IEEE Computer Society, Washington (2007)
5. Ennals, R.: Software transactional memory should not be obstruction-Free. Technical report, Intel Research Cambridge (2006)
6. Felleisen, M.: The theory and practice of first-class prompts. In: POPL 1988, pp. 180–190 (1988)
7. Flanagan, C., Freund, S.N.: Atomizer: a dynamic atomicity checker for multithreaded programs. In: POPL 2004, pp. 256–267. ACM (2004)
8. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: PLDI 1998, pp. 212–223 (1998)
9. Harris, T., Larus, J., Rajwar, R.: Transactional Memory, 2nd edn. Morgan and Claypool Publishers, San Rafael (2010)
10. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: ISCA 1993, pp. 289–300. ACM Press (1993)
11. Hicks, M., Foster, J.S., Pratikakis, P.: Lock inference for atomic sections. In: TRANSACT 2006, Ottawa, Canada, May 2006, pp. 95–102 (2006)
12. Imam, S., Sarkar, V.: Habanero-java library: a java 8 framework for multicore programming. In: PPPJ 2014, pp. 75–86. ACM (2014)
13. Lock (Java platform SE 7), September 2014. http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html
14. Larus, J., Kozyrakis, C.: Transactional memory. Commun. ACM 51(7), 1364800, 80–88 (2008)
15. Lublinerman, R., Zhao, J., Budimlić, Z., Chaudhuri, S., Sarkar, V.: Delegated isolation. In: OOPSLA 2011, pp. 885–902 (2011)
16. Martin, M., Blundell, C., Lewis, E.: Subtleties of transactional memory atomicity semantics. IEEE Comput. Archit. Lett. **5**(2), 17–17 (2006)
17. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: stanford transactional applications for multi-processing. In: IISWC, pp. 35–46. IEEE (2008)
18. Multiverse: software transactional memory for Java and the JVM, April 2012. http://multiverse.codehaus.org/overview.html
19. Nguyen, D., Lenharth, A., Pingali, K.: A lightweight infrastructure for graph analytics. In: SOSP 2013, pp. 456–471. ACM (2013)
20. OpenMP API, version 4.0, July 2013. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
21. Pingali, K., et al.: The tao of parallelism in algorithms. In: PLDI 2011, pp. 12–25. ACM (2011)
22. Rajwar, R., Goodman, J.R.: Speculative lock elision: enabling highly concurrent multithreaded execution. MICRO **34**, 294–305 (2001)
23. Srinivasan, S., Mycroft, A.: Kilim: isolation-typed actors for java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 104–128. Springer, Heidelberg (2008)
24. Stadler, L., Wimmer, C., Würthinger, T., Mössenböck, H., Rose, J.: Lazy continuations for java virtual machines. In: PPPJ 2009, pp. 143–152 (2009)