7th International Symposium on
High-Level Parallel Programming and Applications (HLPP 2014)
Amsterdam, Netherlands
July 3-4, 2014

Atomic RMI: a Distributed Transactional Memory Framework

Konrad Siek · Paweł T. Wojciechowski

Abstract Multiprocessor programming is an art: parallel execution can cause operations executed by separate processes to interweave in unexpected ways and produce incorrect results. The same problems carry over to distributed environments. Concurrent execution can be controlled with low-level mechanisms such as distributed barriers, locks, and semaphores. However, they are notoriously difficult and cumbersome tools to use. Therefore, new approaches are being researched to make both multiprocessor and distributed concurrency both more transparent and less error-prone. *Transactional Memory* is one such approach where programmers designate transactions and the system transparently solves concurrency and other (e.g. fault tolerance) problems. This paper presents Atomic RMI, a framework that implements a distributed transactional memory framework with fault tolerance mechanisms. It extends Java RMI with a pessimistic concurrency control algorithm that provides exclusive access to shared objects and supports rollback and fault tolerance. It has an automatically triggered early-release of objects to improve efficiency. It also allows any operations within transactions, including irrevocable ones, like system calls, and provides an unobtrusive API. Our evaluation shows that Atomic RMI performs better than similarly-implemented fine grained locking mechanisms in most distributed environments. Atomic RMI also performs better than an optimistic TM in environments with high contention and a high ratio of write operations, while being competitive otherwise.

Keywords Concurrency Control · Distributed Systems · Software Transactional Memory

1 Introduction

When programmers want to increase their systems' performance, or make them more reliable, they increasingly turn to parallel and distributed computing.

K. Siek · P. T. Wojciechowski
Institute of Computing Science, Poznań University of Technology
E-mail: konrad.siek@cs.put.edu.pl, pawel.t.wojciechowski@cs.put.edu.pl

Using a multiprocessor system can increase throughput by allowing some parts of code to compute independently on different processors and join only to perform synchronization as necessary. The same is true for using a distributed computing environment, where a complex task can be distributed so that every node, each an independent multiprocessor environment, can perform some part of task in parallel. Moreover, a well-designed network is a more robust system that can be made to withstand crashes of any single node, without completely stopping. Given these advantages, many emerging applications, like simulation-driven drug discovery or social network analysis, can only be achieved through the combined effort of tens of thousands of servers acting as a single warehouse-scale computer.

However, parallel execution notoriously can cause operations on separate nodes to interweave in unexpected ways and produce wrong (i.e.,. unexpected and unintuitive) results. For example, a node executing a series of operations on shared data may find that another node modified the same data in the meantime, causing the system to become inconsistent. In addition, distributed systems must solve problems like e.g., partial failures and lack of global coordination. The programmer must consider the possible problems and deal with them manually. This means ensuring atomicity, consistency, and fault tolerance, using mechanisms like distributed barriers, locks, and semaphores. However, such low-level mechanisms are notoriously difficult to use correctly, since one must reason about interleavings of seemingly irrelevant parts of distributed systems. They also obscure code with concurrency control instructions.

Given the required expertise, many programmers seek recourse in simple solutions, like a single global lock for all shared data accesses. While such techniques are simple to use, they severely limit the level of parallelism achievable in the system. They also introduce bottlenecks in distributed settings, which prevent the system from scaling—operating equally efficiently as the number of nodes in the network increases.

Consequently, researchers seek ways to make concurrency control more automated and easier, while retaining a decent level of efficiency. *Transactional Memory* (TM) [9,20] is one such approach, where the programmers use the transaction abstraction to annotate blocks of code that must be executed giving the illusion of being executed sequentially (e.g., ensuring serializability [16]). The TM system then ensures this is the case using an underlying concurrency control algorithm, whose details may remain hidden from the programmer. In effect, transactions make it easier for conventionally-trained software engineers to reason about the state of the distributed system, and so, reduce the effort required to implement correct systems. In addition, the underlying concurrency control algorithm can ensure a decent level of parallelism.

The TM approach can be applied to distributed system as well as multiprocessor ones, although additional problems, like partial failures, need to be addressed by *distributed TM*. On the other hand, distributed TM also presents new opportunities. In non-distributed TM operations perform reads and writes on shared data. However, a distributed TM can also allow transactions to execute code on remote nodes. In effect, transactions in Distributed TM can

become distributed transactions and execute in part on different machines in the network. This model is referred to as the *control flow* model [1] (as opposed to the read-write only *data flow* model [23]).

In TM emphasis is placed on optimistic concurrency control. There are variations, but generally speaking in this approach a transaction executes regardless of other transactions and performs validation only when it finishes executing (at commit-time). If two transactions try to access the same object, and one of them writes to it, they conflict and one of them aborts and restarts. When a transaction aborts, it should not change the system state, so aborting transactions must revert the objects they modified to a checkpoint. Alternatively, they work on local copies and merge them with the original object on a successful commit.

Unfortunately, there is a problem with irrevocable operations in the optimistic approach. Such operations as system calls, I/O operations, or network messages, once executed, cannot be canceled and so, cause aborted transactions to have a visible effect on the system. In a distributed context these operations are common. The problem was avoided by using irrevocable transactions that run sequentially, and so cannot abort [26], or providing multiple versions of transaction view for reads [2,18]. In other cases, irrevocable operations are just forbidden in transactions (e.g., in Haskell).

A different approach, as suggested by [14] and our earlier work [27,28], is to use fully-pessimistic concurrency control. This involves transactions waiting until they have permission to access shared objects. In effect, conflicting operations are postponed and transactions, for the most part, avoid forced aborts. And therefore, transactions naturally avoid the problems stemming from irrevocable operations.

This paper presents *Atomic RMI*, a programming framework that extends Java RMI with an implementation of a distributed TM in the control flow model. Atomic RMI uses the *Supremum Versioning Algorithm* (SVA) [22] as the underlying concurrency control algorithm. SVA is a fully-distributed pessimistic TM algorithm with support for programmatic rollback and an early-release mechanism, which allows transactions to hand over shared objects to other transactions, in certain situations even before the original owner commits.

In the paper, we give a broad overview of our system in Section 2 and of SVA in Section 3, followed by a discussion of strengths and limitation of Atomic RMI in Section 4 and a comparison to other work in the field in Section 6. Finally, we present the results of an experimental evaluation in Section 5, where we show the gain in efficiency compared to some typical locking approaches and an optimistic distributed TM.

2 Overview

Atomic RMI works in a distributed environment of *Java Virtual Machines* (JVMs), as depicted in Fig. 1. Atomic RMI architecture is strongly based on

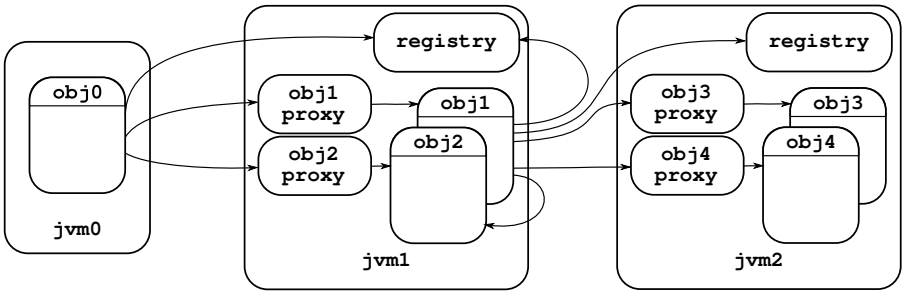

Fig. 1: Atomic RMI architecture.

the architecture of Java RMI. Each node can host a number of shared remote objects, each of which is registered in an RMI registry located on the same node. Each remote object specifies an interface of methods that can be called remotely. A client application running on any JVM can ask any registry for a reference to a particular object. Then, the client can use the reference to call the object methods. Each method's code then executes on the object's host node and returns the result to the client.

Atomic RMI introduces transaction-based concurrency control to this model. Clients calling remote objects' methods do so within clearly-defined transactions, and the system makes sure that the transactions are executed correctly. Atomic RMI ensures correctness using SVA, a TM algorithm (see Section 3).

In order to give SVA the means to guide execution so that correctness is guaranteed, Atomic RMI introduces remote object proxies into the RMI architecture. For each shared remote object there is an automatically-generated proxy on the host node that has a wrapper method for each of the original object's methods (those available remotely). Clients are required to access remote objects via proxies, so all calls of the original object's methods first pass through wrapper methods. The wrapper methods are then used to enforce SVA: establish whether a given operation can be executed at a given time, and deffer or cancel them if necessary. Once the wrapper establishes that a call may proceed, the proxy calls the original method of the remote object.

On the client-side, Atomic RMI requires that blocks of code that access remote objects must be designated as transactions, if correctness is to be ensured. An example of transactional code in Atomic RMI is given in Fig. 2. The example shows a transaction which withdraws a sum of `100` from one account and deposits it on another. The transaction only succeeds if the first account does not become overdrawn, otherwise its effects are erased.

To designate a transaction, the programmer creates a transaction object (line 1 in Fig. 2) and calls its `start` (line 4) and `commit` (line 8) methods to indicate where the transaction begins and ends. Alternatively, the transaction may abort at the end using the `rollback` method (line 10)—then all the effects of the transaction on the system are reverted. Rollback can be used to facilitate application logic or handle exceptions and errors.

```
1  Transaction t = new Transaction(...);
2  a = t.accesses(registry.lookup("A"), 2);
3  b = t.accesses(registry.lookup("B"), 1);
4  t.start();

5  a.withdraw(100);
6  b.deposit(100);

7  if (a.getBalance() > 0)
8      t.commit();
9  else
10     t.rollback();
```

Fig. 2: Atomic RMI transaction example.

The code between transaction's start and commit (or abort) is its *body* (lines 4–10). The body of an Atomic RMI transaction can contain local operations as well as method calls to remote objects. Any code executed as part of remote method execution within a transaction is also part of the transaction. So Atomic RMI transactions are *distributed transactions*: the execution of their code can be distributed among several nodes.

The code executed prior to start that sets up the transaction is called it's *preamble*. Atomic RMI preambles triggers the creation of remote proxy objects and provides the information for SVA's early release mechanism (see Fig. 4.1). The programmer is required to use the `accesses` method of the transaction object to indicate which remote objects can be used within the transaction (lines 2–3). This prompts the host of the remote object to generate a proxy object—a separate proxy is generated for each transaction. The `accesses` method returns a reference to the proxy object which is then used within the transaction to call the remote object's methods. The method also allows the maximum number of method calls on each remote object to be declared—this allows Atomic RMI to increase the efficiency of transaction execution through the early release mechanism.

On the server-side, shared remote objects used with Atomic RMI are plain unicast (stateful) RMI objects, except that instead of `UnicastRemote- Object` (which provides Java Remote Method Protocol handling) they subclass the `TransactionalUnicastRemoteObject` class. This class creates proxy objects when necessary, in effect injecting SVA support code into remote method invocations. The methods of remote objects are not limited: as well as simple operations like reading and writing to a field, they can contain blocks of code which include side effects, system calls, I/O operations, network communication etc. that execute on the server. This freedom is possible in large part to the pessimistic approach to concurrency control used by SVA—since these operations often produce visible effects on the system, they cannot be repeated in case of conflicts, as in the optimistic approach. The pessimistic approach will only let them execute (up to) once in the course of normal operation, although allowances must be made when the user manually triggers an abort by rolling back some transaction.

In particular, remote methods can also contain method calls to other remote objects, further distributing the execution of the transaction. Note however, that if these are to be accessed transactionally (i.e., with the same correctness guarantees), the calls have to be included in the upper bounds defined in the transaction's preamble.

Note that Atomic RMI uses the control flow model of execution, since shared objects allow transactions to execute code on remote objects, rather than limiting them to reading and writing data, as in the data flow model. Our intention is to orientate Atomic RMI towards this model, since it provides greater freedom and expressiveness to the programmer, who can balance the load between servers and clients by defining the level of processing that is done on remote objects. Additionally, the control flow model is more versatile, because it can emulate the data flow model if remote objects support methods which simply write or retrieve data from the host.

3 Supremum Versioning Algorithm

The underlying concurrency control of Atomic RMI is implemented using the *Supremum Versioning Algorithm* (SVA) with rollback support [29,22]. SVA is a transactional concurrency control algorithm. That is, blocks of code are executed as atomic transactions. The algorithm guarantees exclusive access to shared objects as long as a transaction requires it (in particular, objects can be released early). More precisely, Atomic RMI guarantees transaction serializability [16]—any concurrent execution of transactions is equivalent to some serial execution of those transactions. In addition Atomic RMI executions are recoverable [8]—a transaction which reads from an earlier transaction will only complete (abort or commit) after the earlier one does. Atomic RMI also preserves transaction real-time–order (see e.g., [7])—any non-concurrent transactions retain their order.

SVA is pessimistic—it delays operations on shared objects, rather than optimistically executing them and rolling transactions back if conflicts appear. Furthermore, synchronization is achieved using a fully distributed mechanism based on version counters associated with individual remote objects and/or transactions. Below we give a rudimentary explanation of the algorithm, but this variant of SVA is described in detail in [22].

SVA uses four version counters to determine whether any transactional action (e.g., executing a method call on a shared object or committing a transaction) can be allowed, or whether it needs to be delayed or countermanded. *Global version counter* `o.gv` counts how many transactions that asked to access object `o` started. *Private version counter* `o.pv[k]` remembers transaction `k`'s version number for object `o`. *Local version counter* `o.lv` stores the version number for `o` of the last transaction which released `o`. *Terminal local version counter* `o.ltv` similarly stores the version number for `o` of the last transaction which released `o`, but only if it already committed or aborted. There is one of each of these counters per remote object, and they are either a part of the

```
for(o : sort(k.sup))
  o.lock.acquire();
for(o : k.sup) {
  o.gv += 1;
  o.pv[k] = o.gv;
}
for(o : sort(k.sup))
  o.lock.release();
```

Fig. 3: Initialization

```
k.wait(o.pv[k]-1 == o.lv);
if (o.isInvalid())
    k.rollback();
o.m(...);  // call method
           // on shared object
k.cc[o] += 1;
if (k.cc[o] == k.sup[o])
  o.lv = o.pv[k];
```

Fig. 4: Method call

```
for (o : k.sup) {
  k.wait(o.pv[k]-1 == o.ltv);
  if (o.isInvalid())
    {k.rollback(); return;}
}
for (o : k.sup) {
  if (k.cc[o] < k.sup[o])
    o.lv = o.pv[k];
  o.ltv = o.pv[k];
}
```

Fig. 5: Commit

```
for (o : k.sup) {
  k.wait(o.pv[k]-1 == o.ltv);
  o.restore(k);
}
```

Fig. 6: Rollback

```
k.wait(o.pv[k]-1 == o.lv);
o.lv = o.pv[k];
```

Fig. 7: Manual release

remote object itself (`o.gv`, `o.lv`, `o.ltv`) or its proxy (`o.pv[k]`). Each remote object also maintains one lock `o.lock` that is used to initialize transactions atomically.

SVA also uses `k.sup`, a map informing which shared objects will be used by transaction `k` and at most how many times each of them will be accessed. Only an upper bound (*supremum*) on the number of accesses is required, and this number can be infinity, if it cannot be predicted. Each transaction also keeps call counters `k.cc[o]` for each object it accesses.

Each transaction's life-cycle consists of an initialization followed by any number of method calls to transactional remote objects, possibly interspersed by local or otherwise non-transactional operations. Finally, it completes by either a commit or abort (rollback) operation. We describe the behavior of SVA with respect to these stages below.

Transaction initialization is shown in Fig. 3. When transaction `k` starts, it increments `gv` for every object in its `sup`, indicating that a new transaction using each of these objects has started. Then, the value of `gv` is assigned to `pv[k]` for each object—the transaction is assigned a unique version of the object that it can access exclusively. For this assignment to be atomic, each transaction must acquire distributed locks (mutexes) for all the objects in `sup` for the duration of its initialization. Note, that this lock is released shortly, and SVA uses other mechanisms for synchronization. Additionally, in order to avoid deadlocks from using fine grained locks, the locks are acquired in a set order. Atomic RMI sorts the locks using their unique identifiers.

After starting, transaction `k` can call methods on shared remote objects (shown in Fig. 4), but only when the object is in the version that was assigned to `k` at start. So, when a method is called on object `o`, `k`'s private version for `pv` is checked against the `o`'s local version `lv`. The local version is the version of

the transaction that released `o` most recently, so if `lv` is one less than `pv`, then the previous transaction released `o`, and `k` can now access `o`. We refer to this condition as the *access condition*. Once it is met, the transaction performs a check to see if the remote object was not invalidated by some other transaction, which released the object early and subsequently aborted. If this happened, the current transaction cannot proceed working on inconsistent data and must also roll back. Otherwise, the transaction performs the actual method call and in effect the code of method `o.m` is executed on the remote host. After transaction `k` calls, the method the call counter `o.cc` is incremented. If the counter is equal to `k`'s maximum declared number of calls to `o`, then object `o` is released by assigning `k`'s private version to `o`'s local version. This means that some other transaction with `pv` one greater than `k` can now also access `o`.

Since SVA is created with the control flow model in mind, it makes no distinction between reads and writes. That is, all method calls to remote objects are treated as potential modifications to the remote object. This means that certain standard optimizations with respect to read-only transactions are not used in Atomic RMI's version of SVA. This is done in trade for the expressiveness and versatility allowed by the control flow model and offset by the early release mechanism, as we show in Section 5.

When transaction `k` finishes successfully, it commits as shown in Fig. 5. First for each object `o` used by `k`, `k` waits until the terminal local counter `ltv` is one less than its own private version `pv`—this means that the previous transaction completes and either committed or aborted. We refer to this condition as the *commit condition*, and it is analogous to the access condition. Passing it is necessary before the transaction can complete, because if any previous transaction from with which `k` shared some object rolled back, then `k` would be working on inconsistent data. So, in order to leave the possibility to force `k` to abort in such situations, all transactions must wait until preceding transaction with which they shared objects commit or rollback. If there was no rollback, the transaction releases all shared objects that it did not release yet (i.e., the upper bound in `sup` was not reached) by assigning `pv` to `lv`. Finally, the transaction indicates that it completed by assigning `pv` to `ltv`.

Alternatively, the programmer may induce a rollback (Fig. 6), or a rollback may be forced during commit or a method call. In that case `k` waits until the previous transaction completed and restores each object `o` to a state prior to the transaction. Restoring the object invalidates it to all subsequent transactions, which causes them to roll back as well. This is much more subtle in practice and involves checkpoint management, but we omit the details as they are, on the whole, secondary matters (see [22] for a full treatment).

In certain situations, the programmer may have a good knowledge of when an object stops being used in a transaction from the semantics of the program. In such cases the programmer can then allow a remote object to be released early and in this simple manner increase the efficiency of the system. For this reason, SVA includes a manual release operation that can be manually invoked, shown in Fig. 7. First, transaction `k` waits for `lv` to reach `pv` minus one for `o`, as with calling a method, because `k` cannot release an object it should not yet

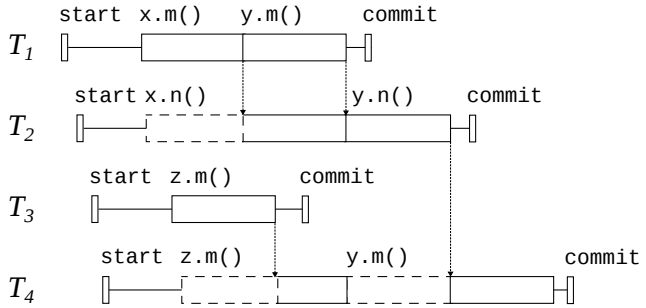

Fig. 8: Parallel execution of SVA transactions.

even have access to. Then `pv` is assigned to `lv`, and another transaction can access it (and `k` cannot).

The early release mechanism makes SVA capable of achieving a relatively high level of parallelism by interweaving pairs of transactions that access the same objects and by making transactions that do not share objects independent of one another. For example, if there are two transactions T_1 and T_3, as in Fig. 8, that use different objects from each other (T_1 uses `x` and `y`, T_3 uses `z`) they do not interfere with each other, so they can run in parallel. However, since transactions T_1 and T_2 access the same objects (`x` and `y`), T_2 can only start using `x` or `y` once T_1 releases them. However, T_2 does not wait with accessing `x` and `y` until T_1 commits. Assuming transactions have precise suprema specified, objects are released as soon as each transaction performs its last operation on them. This way T_1 and T_2 can run partially in parallel, and therefore T_1 and T_2 together finish execution faster than they would if they were run sequentially. Similarly, T_4 must wait for T_3 to release `z` and for T_2 to release `b`, but it can execute code using `z` in parallel to T_2 executing its own operations on other objects.

4 In-depth Look

In this section we discuss in greater detail some of the mechanisms employed in Atomic RMI and their implications.

4.1 Suprema

The main requirement that Atomic RMI poses for its users is the need to provide the set of objects used by transactions *a priori* and a strong suggestion to also provide upper bounds (suprema) on the number of accesses of remote objects accessed by each transaction. The former is required to acquire versions on each object. The latter allows Atomic RMI to decide when objects can be released early—if this information is inexact or omitted (equivalent to setting the upper bound to infinity) Atomic RMI will only release objects

when transactions commit or roll back, and forgo early release. In such cases Atomic RMI will be less efficient (transactions will wait more on one another), although the execution will nevertheless be correct.

However, while it is acceptable for upper bounds to be too high, it is essential that they are never lower than the actual number of calls a transaction does to a given object. If the specification is lower then the actual number of accesses, the guarantees provided by SVA cannot be upheld, because a transaction could release an object and then attempt to access it again. Because when transaction `k` releases object `o`, counter `o.lv` is set to `o.pv[k]`, then access condition `o.pv[k] - 1 = o.lv` is no longer met for `k`, so the transaction will not be able to access `o` again. And since transaction versions in are unique, no other transaction will be able to set `o.lv`, to value that would allow `k` to access `o` again and would perpetually wait at the access condition for `o`. In order to alleviate such situations, transactions throw an exception when the number of accesses for some object exceeds its supremum. It is then left to the programmer to resolve the issue by handling the exception. A typical solution would be to roll back the offending transaction. A more sophisticated technique would be to roll back the transaction, then modify the suprema, and retry.

The upper bounds can be collected manually by the programmer by inspecting the code and creating the preamble. They can also be inferred automatically by various means, including a type system (e.g., [27]) or static analysis. Atomic RMI comes with a precompiler tool which statically analyses transactions to discover which objects they use, and to derive the upper bounds on accesses to them. With this information, the precompiler generates the appropriate code and inserts it into the program. The idea behind the static analysis is described in [21]. The tool itself is a command-line utility implemented on top of the Soot framework [25].

4.2 Manual Early Release

The early release mechanism in Atomic RMI can be triggered automatically or manually. The two methods complement each other.

Note the simple example in Fig. 9, where a transaction calls methods on shared objects `a` and `b` in a loop. If manual release was to be used, the simplest way to use it is to inserted release instructions at the end of the loop at lines 9–10. However, it will mean that before `a` is released, the transaction unnecessarily waits until `b` executes as well. If `a` and `b` are remote objects, each such call can take a long time, so this simple technique impairs efficiency.

Instead, the programmer should strive to write transactions like in Fig. 10. Here, `a` is released at lines 7–8, in the last iteration of the loop before the method call on `b` is started. An earlier release improves parallelism, but the solution requires that the programmer spends time on optimizing concurrency (which the TM approach should avoid) and clutters up the code with instructions irrelevant to the application logic. In addition, the release in both

```
1  t = new Transaction(...)
2  a = t.accesses(a);
3  b = t.accesses(b);
4  t.start();

5  for (i = 0; i < n; i++) {
6    a.run();
7    b.run();
8  }
9  t.release(a);
10 t.release(b);


11 // local operations
12 t.commit();
```

Fig. 9: Early release at end of block.

```
1  t = new Transaction(...)
2  a = t.accesses(a);
3  b = t.accesses(b);
4  t.start();

5  for (i = 0; i < n; i++) {
6    a.run();
7    if (i == n)
8        a.release();
9    b.run()
10 }
11 t.release(b);

12 // local operations
13 t.commit();
```

Fig. 10: Conditional early release.

```
1  t = new Transaction(...)
2  a = t.accesses(a, n);
3  b = t.accesses(b, n);
4  t.start();

5  for (i = 0; i < n; i++) {
6    a.run(); // nth call: release
7    b.run(); // nth call: release
8  }

9  // local operations
10 t.commit();
```

Fig. 11: Early release by supremum.

```
1  t = new Transaction(...)
2  for (h : hotels)
3      h = t.accesses(h, 2);
4  t.start();

5  for (h : hotels) {
6      if (h.hasVacancies())
7          h.bookRoom();
8      else
9          t.release(h);
10 }
11 t.commit();
```

Fig. 12: Complementary manual release.

examples sends an additional network message to `a` and `b` (because the release method requires it), which can be relatively expensive.

However, if the algorithm is given the maximum number of times each object is accessed by the transaction, i.e., that `a` and `b` will be accessed at most `n` times each, then Atomic RMI can determine which access is the last one as it is happening. Then, the transaction's code looks like the example in Fig. 11, where suprema are specified in lines 2–3, but the instructions to release objects are hidden from the programmer, so there is no need for supplementary code. Additionally, since release is done as part of the `n`th execution of a call on each object, there is no additional network traffic. Furthermore, `a` does not wait for `b` to execute.

However, releasing by suprema alone is not always the best solution, since there are scenarios when deriving precise suprema is impossible. In those cases the manual early release complements the suprema-based mechanism. One such case is shown in Fig. 12, where a transaction searches through objects representing hotels, and books a room if there are vacancies. So each interaction with a hotel can take up to two method calls: vacancy check (line 6) and booking (line 7). However the supremum will only be precise for one hotel, the first one with vacancies. Other hotels that do not have vacancies, will not be asked to book room, so there is only one access. This means that the supre-

mum will not be met for those cases until the end of the transaction, so they will be released only on commit. Hence, they are manually released on line 9, so they can be accessed by other transactions as soon as possible.

4.3 Irrevocable Operations

The greatest advantage of Atomic RMI is its novel pessimistic algorithm, which allows any operation to be used within transactions. In particular, irrevocable operations pose no problem. These are operations that have visible effects on the system and cannot be easily reverted, e.g., system calls, sending network messages. This is not true for optimistic transactions, because conflicts cause rollbacks, which then cause irrevocable transactions to be repeated.

For the same reason, Atomic RMI allows transactions to include locking or to start new threads within transactions. This is also often not possible in optimistic transactions, where rollbacks can cause too many threads to be started or locks to be acquired and not released (especially, if conflicts are detected eagerly). However, not only does allowing these sorts of operations improve expressiveness, but it also makes working with legacy code easier.

4.4 Nesting and Recurrency

Atomic RMI supports transaction nesting, albeit with limitations. The programmer can create a transaction within another transaction, but in such cases it is vital to ensure that they do not share objects. Otherwise, the inner transaction will wait for the outer to release the objects, while the outer will not release them until the inner finishes. In effect, a deadlock occurs (although, in the original SVA [29] nesting is not an issue since all transactions are parallel).

Atomic RMI also supports transaction recurrency. That is, a transaction may call itself within itself. Atomic RMI provides an interface called `Transactable` that allows transactions to be enclosed within a method, rather than between `start` and `commit`, and the method may then be used recurrently. The recursion will be treated as a single transaction. The execution will proceed until the methods `commit`, `rollback`. are called, in which case the transactional method is exited and the transaction finishes as normal. Keep in mind, however, that the suprema for object accesses must still be defined for the entire execution of the transaction.

4.5 Fault Tolerance

In distributed environments detecting and tolerating faults of network nodes is vital. Atomic RMI can suffer two basic types of failures: remote object failures and transaction failures.

Remote objects failures are straightforward and the responsibility for detecting them and alarming Atomic RMI falls onto the mechanisms built into

Java RMI. Whenever a remote object is called from a transaction and it cannot be reached, it is assumed that this object has suffered a failure and an exception is thrown. The programmer may then choose to handle the exception by, for example, re-running the transaction, or compensating for the failure. Remote object failures follow a *crash-stop* model: any object that crashed is removed from the system.

Transaction failures are those where remote objects accessed by a transaction loose communication with the transaction (via timeout). In that case the transaction (or more likely the client application) is considered to have crashed. Therefore, the affected remote objects revert to the state immediately prior to the start of the transaction, so that all changes done by the failed transaction are forgotten. Then, the objects are released, so other transactions may use them. Note that, since SVA will update the versions of the objects the old transaction cannot then access them again, if failure detection were imperfect.

5 Evaluation

In this section we present the results of a practical evaluation of Atomic RMI. First, we compare the performance of Atomic RMI to other distributed concurrency control mechanisms, including another distributed TM. In the second test, we check the performance of Atomic RMI under different Java Runtime Environments (JREs).

Benchmarks For our comprehensive evaluation we used a micro-benchmark and three complex benchmarks. We based our implementation of the benchmarks on the one included in HyFlow [19].

The *distributed hash table benchmark* (DHT) is a micro-benchmark containing a number of server nodes acting as a distributed key-value store. Each node is responsible for storing values for a slice of the key range. There are two types of transactions executed by clients in the system. A write transaction selects 2 nodes and atomically performs a write on each. A read transaction selects 4 nodes and performs an atomic read on them. The benchmark is characterized by small transactions (2-4 operations, few local operations) and low contention (few transactions try to access the same resource simultaneously).

The *bank benchmark* simulates a straightforward distributed application using the bank metaphor. Each node hosts a number of bank accounts that can be accessed remotely by clients. Bank accounts allow write operations (*withdraw* and *deposit*) and a read operation (*get balance*). Clients perform either write or read transactions. In the former type, a transfer transaction, two random accounts are selected and some sum is withdrawn from one account and deposited on the other. In the latter type, an audit transaction, all the accounts in the bank are atomically read by the transaction and a total is produced. The benchmark has both short and long transaction and medium to high contention, depending on the number of read-only transactions.

The *loan benchmark* presents a more complex distributed application where the execution of the transaction is also distributed. Each server hosts a number of remote objects that allow write and read operations. Each client transaction atomically executes two reads or two writes on two objects. When a read or write method is invoked on a remote object, then it also executes two reads or writes (respectively) on two other remote objects. This recursion continues until it reaches a depth of five. Thus, each client transaction "propagates" through the network and performs 30 operations on various objects. Hence, the benchmark is characterized by long transactions and high contention, as well as relatively high network congestion.

Finally, the *vacation benchmark* is a complex benchmark (originally a part of STAMP [15]), representing a distributed application with the theme of a travel agency. Each server node supplies three types of objects: cars, rooms, and flights. Each of these represents a pool of resources that can be checked, reserved, or released by a client. When some resource is reserved, associated reservation and customer objects are also created on the server. Clients perform one of three types of transactions. *Update tables* selects a number of random objects and changes their price to a new value. *Delete customer* removes a random customer object along with any associated reservations. This transaction may require programmatic use of rollback. *Make reservation* is a read-dominated transaction that searches through a number of objects, chooses one of each type (car, room, flight) that meet some price criterion. Once the objects are chosen, the transaction may create a reservation. The benchmark has medium to large transactions with a lot of variety, and medium to high contention.

Frameworks We evaluate Atomic RMI with specified precise suprema (where possible). All versions of Atomic RMI use manual early release in the vacation benchmark to improve efficiency while making reservations (while searching through remote objects). In all other cases transactions release objects when they reach their supremum.

We used two lock-based distributed concurrency control mechanisms for comparison, used as transactions: standard Java RMI with exclusion locks (denoted *RMI Locks*) and Java RMI with read/write locks (*RMI R/W Locks*). These mechanisms were chosen because they are similarly easy to use as Atomic RMI. They feature fine grained locking: there is one lock per remote object. The locks are used within a transaction abstraction, such that at the start of the transaction all locks are acquired (sorted, to avoid deadlock), and all the locks are released on commit. We use exclusion locks as a baseline algorithm, which is very simple to use and can be expected to be seen in applications written by conventionally trained software engineers. On the other hand R/W lock present one of the most popular types of performance optimizations in concurrent systems: parallelizing reads.

We also compare Atomic RMI with HyFlow [19], another Java RMI-based implementation of transactional memory. Specifically, we use Distributed Transactional Locking II (DTL2), HyFlow's distributed variant of TL2 [6], a well-

known optimistic TM. Since the technology used in both Atomic RMI and HyFlow is the same, the comparison should show the performance difference between the pessimistic and optimistic approaches to TM.

Note that the delete customer operation in the Vacation benchmark requires some transactions to execute speculatively and abort when the list of objects reserved for deletion becomes out of date. In that case Atomic RMI and HyFlow transactions use the rollback operation. However, the lock-based frameworks we use do not have rollback support, so an *ersatz* rollback mechanism must be implemented within these transactions.

Testing Environment In each of the benchmarks every node performs the rôle of a server hosting a number of publicly accessible remote objects, as well as a client running various randomly chosen types of transactions using remote objects from any server.

We perform our tests on a 10-node cluster connected by a private 1 Gb network. Each node is equipped with two quad-core Intel Xeon L3260 processors at 2.83 GHz with 4 GB of RAM each and runs a OpenSUSE 13.1 (kernel 3.11.10, x86_64 architecture). We use the 64-bit IcedTea 2.4.4 OpenJDK 1.7.0_51 Java runtime (suse-24.13.5-x86_64) for tests involving comparison between multiple frameworks. We also use this JRE (denoted OpenJDK 1.7.0_51 on the graphs) alongside Oracle's 64-bit 1.7.0_55-b13 Java Runtime Environment, Java HotSpot build 24.55-b03 (denoted Oracle 1.7.0_55); and Oracle's 64-bit 1.8.0_05-b13 Java Runtime Environment, Java Hotspot build 25.5-b02 (denoted Oracle 1.8.0_05) for evaluating behavior when running on different JREs. We also attempted to run the benchmarks on the latest version of Oracle JRockit (1.6.0_45-b06), but were unsuccessful due to compatibility issues with the libraries used for the implementation of the frameworks and benchmarks, most notably, Deuce STM instrumentation [12].

Each of the benchmarks is run on 2–10 nodes. Every node hosts one server with as many objects as specified by the benchmark. In addition every node hosts one client with 24 threads each. So, for example, on 10 nodes there are 240 simultaneous transactions accessing objects on 10 nodes. Threads execute transactions selected at random. In one batch of tests there are 20% of read transactions and 80% of write transactions in each benchmark. In the other batch the ratio is reversed.

Results The results of the comparison between concurrency control mechanisms are presented in Fig. 13.

The result of the DHT benchmark show that in a low-contention environment with short transactions Atomic RMI is comparable to performance obtained by using both fine grained R/W locks and HyFlow, and all three are much better than fine grained exclusion locks. Atomic RMI's advantage over exclusion locks comes from early release and allowing some transactions to execute in part in parallel. Thus, there are more transactions executing at once, so more of them can go through the system per a unit of time. R/W locks and HyFlow attain a very similar result, by allowing reads to execute in

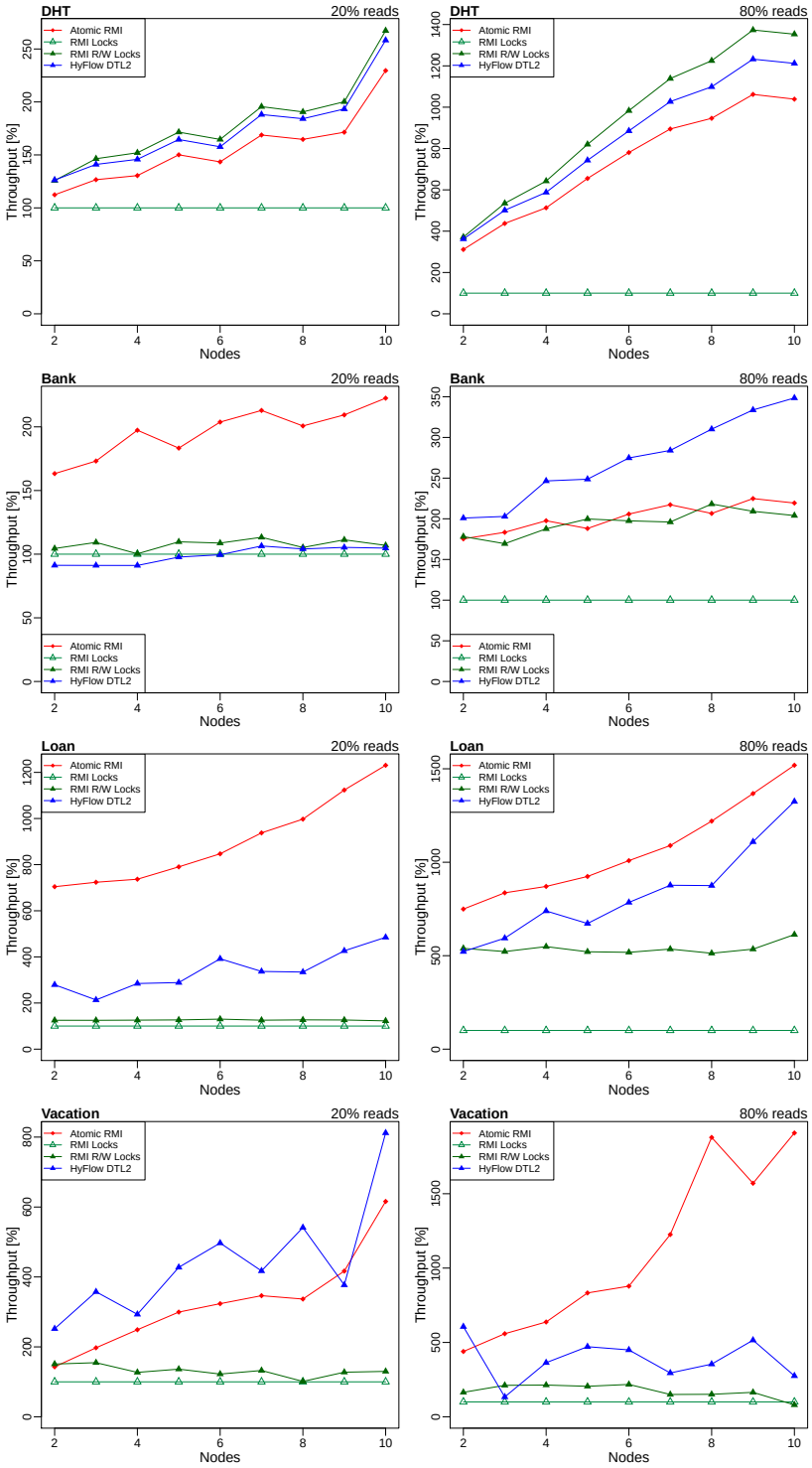

Fig. 13: Evaluation results by benchmark (DHT, Bank, Loan, and Vacation) and read/write transaction ratio (20% reads and 80% reads). Each benchmark is presented on two graphs: one for 20–80 read/write operation ratio, and the other for 80–20 ratio. Points on the graph represent the mean throughput (on the y-axis) from the given benchmark run on a particular number of nodes (on the x-axis). The results are shown as a percentage improvement in relation to the execution of RMI Locks.

parallel with other reads, therefore also allowing some transactions to execute in part in parallel. The gain from treating reads specially is very similar to what is gained from early release, so the shapes of the graphs are very similar. However, the overhead of maintaining all of the distributed TM mechanisms in Atomic RMI and HyFlow—including rollback support (so making copies of objects) and fault tolerance (extra network communication)—is greater than the overhead of R/W Locks. Hence, Atomic RMI and HyFlow perform consistently worse in DHT than R/W Locks. Note also that the advantage that the more subtle frameworks have over exclusion locks is growing with time, hinting at better scalability.

The results for Bank show a case with higher contention. Here, if R/W Locks and HyFlow can take advantage of a large number of read-only transactions, they can do similarly as well as Atomic RMI in the case of the former, or better in the case of the latter. However, since contention is higher here, the higher cost of setting up HyFlow's and Atomic RMI's more complex concurrency control pays off, so both framework tend to outdo R/W Locks on average. Since read-only transactions here contain operations in random order, Atomic RMI's versioning algorithm is often forced to wait for a preceding transaction to release the right object. In effect, it has trouble allowing large sections of transaction to run in parallel with other transactions. However, in HyFlow any two read-only transactions can operate almost completely in parallel. This shows, that Atomic RMI would greatly benefit from introducing support for read/write operation differentiation. On the other hand, in a scenario where read/write transactions are the norm, Atomic RMI significantly outperforms all other benchmarks, since there is nothing to gain from distinguishing between operations. In such a case, R/W Locks act almost identically to Exclusion Locks. In this case, HyFlow allows for some parallelism bu falls prey to a high number of aborts caused by speculative execution of write operations. Here, HyFlow transactions abort in between 15.5% and 51% cases, as opposed to between 4.25% and 8.9% in the previous scenario. Note that other transactions do not perform aborts in this scenario at all. Note also, that, since Atomic RMI does not distinguish between reads and writes, as can be expected, performance is more-or-less constant between the scenarios.

The Loan benchmark shows that Atomic RMI is also much better at handling long transactions and high contention than all other types of the concurrency control mechanisms. Since Atomic RMI does not distinguish between reads and writes, both scenarios are effectively the same in terms of performance. Again, the performance is gained form releasing objects early (while about half of the transaction still remains to be executed) and executing a transaction in parallel in part to the transaction preceding it, and in part to the one following it. Again, R/W Locks and HyFlow perform better in an environment with more read-only transactions than in the one with less of them, but in higher contention, their advantage over Exclusion Locks is not as great as Atomic RMI's.

Finally, Vacation shows the behavior of different types of more complex transactions and in a high contention environment. In this scenario there is

an actual advantage to being able to roll back, and this is a component in the performance advantage. Atomic RMI makes copies for rollback on the server-side, so there is no network overhead associated with either making a checkpoint or reverting objects to an earlier copy. On the other hand, the *ersatz* rollback mechanism requires that clients copy objects and store them on the client side, which makes them operations. Atomic RMI, therefore, has a big advantage over both locking mechanism, from rollback support alone in executing transactions that require them—i.e., *delete customer*. These make up for 10% and 40% of transactions in Vacation (for 80% and 20% percent read transactions percentage, respectively). Additionally, the complexity of these real-world–like transactions makes read-only transactions increasingly difficult to parallelize, so R/W Locks struggle with performance, even falling behind exclusion locks at times.

The comparison of HyFlow and Atomic RMI in Vacation is much more involved. First of all, since read transactions do not imply read-only transactions here, there is much less to be gained here by just treating reads specially. Here, even a read transaction can write, so cause conflicts and therefore effect aborts in HyFlow. It is the order of operations and the implementation of the read transaction that explains why Atomic RMI does better than HyFlow in the 20% read scenario. First of all, the *make reservation* transaction initially performs a sequence of reads to a large set of remote objects, until one is found that fits some specified criterion. Since the object can be released instantly if the criterion is not met, then Atomic RMI allows many parallel transactions to work on the same objects. And since reads in Vacation are done in the same order in each transactions, any two Atomic RMI transactions can execute almost entirely in parallel. On the other hand, since there are writes at the end of *make reservation*, and therefore possible conflicts, HyFlow does not have as much an advantage when executing these transactions. This is why there is the advantage of Atomic RMI over HyFlow in an 80% read scenario. On the other hand, the necessary rollback in *delete customer* is more problematic in Atomic RMI, since it may cause a cascade of rollbacks. Furthermore, *update tables* performs reads in random order, so Atomic RMI encounters the same problems as in Bank. HyFlow avoids these problems through speculation, so it performs better in the 20% scenario.

On the whole, Atomic RMI is able to perform just as well as fine grained locks in all environments, with only small penalty for additional features in environments particularly hostile to versioning algorithms (low contention, short transactions). On the other hand, in environments for which versioning algorithms were intended (high contention, long transactions, mixed reads and writes) Atomic RMI gains a significant performance advantage over fine grained locking. In comparison to HyFlow, both TM systems perform variously in different environments. On average, Atomic RMI tends to perform better than HyFlow in high contention, while it tends to be outperformed by HyFlow in cases where read-only transactions can be treated specially.

The results of the comparison between concurrency control mechanisms are presented in Fig. 14. The benchmarks indicate that Atomic RMI performs in

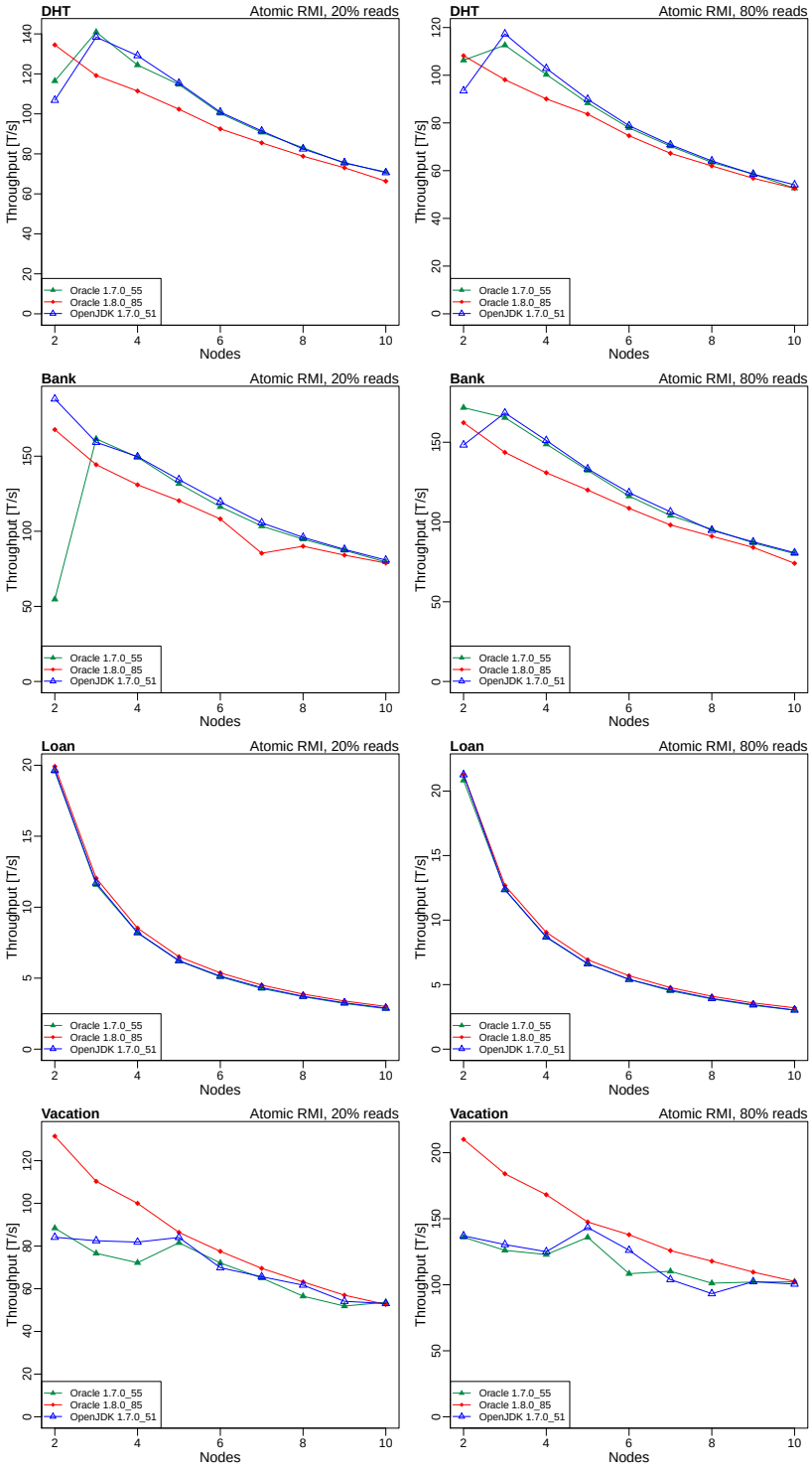

Fig. 14: Evaluation results of Atomic RMI under various JREs. Points on the graph represent the mean throughput in transactions per second (on the y-axis) from the given benchmark run on a particular number of nodes (on the x-axis).

a relatively similar manner. The most significant difference can be seen when relatively few nodes are involved. This is best visible in Vacation for tests with 4 nodes or fewer, where Oracle 1.8.0_85 significantly outperforms either of the Java 7 implementations. The results also show a decline in throughput as more nodes are added. This is because each added node increases the rate of conflicts between transactions, as well as network congestion.

6 Related Work

Atomic RMI is similar to HyFlow [19]. Both use Java RMI as their basis, both support distributed transactions and both allow remote code execution. However, HyFlow uses optimistic concurrency, which—contrary to our approach—incurs inadvertent rollbacks and, in effect, causes problems with irrevocable operations. On the other hand HyFlow support both control flow and data flow execution models. HyFlow2 [24] is an improved version of HyFlow, written in Scala and with advanced nesting support.

Distributed transactions are successfully used where requirements for strong consistency meet wide-area distribution, e.g., in Google's Percolator [17] and Spanner [4]. Percolator supports multi-row, ACID-compliant, pessimistic database transactions that guarantee snapshot isolation. A drawback in comparison to our approach is that writes must follow reads. Spanner provides semi-relational replicated tables with general purpose distributed transactions. It uses real-time clocks and Paxos to guarantee consistent reads. Spanner defers commitment like SVA, but buffers writes and aborts on conflict. Irrevocable operations are completely forbidden in Spanner.

Several distributed Transactional Memory (TM) systems were proposed (see e.g., [3,5,13,11,10]). Most of them replicate a non-distributed TM on many nodes and guarantee consistency of replicas. This model is different from the distributed transactions we use, and has different applications. Other systems extend non-distributed TMs with a communication layer, e.g., DiSTM [13] extends [5] with distributed coherence protocols.

7 Conclusions

We presented Atomic RMI, a programming framework for distributed transactional concurrency control for Java. The transactional method is easy for programmers to use, while hiding complex synchronization mechanisms under the hood. We use that to full effect by employing SVA (with rollback support), an algorithm based on solid theory that allows high parallelism of execution.

Additionally, the pessimistic approach that is used in the underlying algorithm allows out system to present fewer restrictions to the programmer with regard to what operations can be included within transactions. Apart from limited transaction nesting, very little else is forbidden within transactions.

Supremum-based early release makes our programming model efficient and relatively burden-free (especially, when static analysis is employed). Upper

bounds on object calls are harder to estimate statically but the effort pays off since they allow to release objects as early as possible in certain cases. Our evaluation showed that due to the early release mechanism, Atomic RMI has a significant performance advantage over fine grained locks employed in the rôle of transactions. Suprema are also safe: when they are not given correctly (or at all) the system only looses efficiency but maintains correctness, and, at worst, throw an exception.

While the framework is young and some aspects will undoubtedly still need to be ironed out, we are confident that Atomic RMI is a basis for programming distributed systems with strong guarantees. Given the results of our evaluation, in our future work we wish to implement a different versioning algorithm in Atomic RMI, which will distinguish between reads and writes, while retaining the early release mechanism. Combining the two optimizations should improve the efficiency of the system even further.

Acknowledgments The project was funded from National Science Centre funds granted by decision No. DEC-2012/06/M/ST6/00463. Early work on Atomic RMI was funded by the Polish Ministry of Science and Higher Education within the European Regional Development Fund, Grant No. POIG.01.03.01-00-008/08. We would like to thank Wojciech Mruczkiewicz, Piotr Kryger, and Mariusz Mamoński for their preliminary work on this project.

References

1. K. Arnold, R. W. Scheifler, J. Waldo, A. Wollrath, R. Scheifler, and B. O'Sullivan. *The Jini Specification*. Addison-Wesley Longman, 1999.
2. H. Attiya and E. Hillel. Single-version STMs Can Be Multi-version Permissive. In *Proceedings of ICDCN'11: the 12th International Conference on Distributed Computing and Networking*, number 6522 in Lecture Notes in Computer Science, pages 83–94, Jan. 2011.
3. R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software Transactional Memory for Large Scale Clusters. In *Proceedings of PPoPP'08: the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2008.
4. J. C. Corbett and et al. Spanner: Google's Globally-Distributed Database. In *Proceedings of OSDI'12: the 10th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2012.
5. M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2STM: Dependable Distributed Software Transactional Memory. In *Proceedings of PRDC'13: the 15th IEEE Pacific Rim International Symposium on Dependable Computing*, Nov. 2009.
6. D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of DISC'06: the 20th International Symposium on Distributed Computing*, Sept. 2006.
7. R. Guerraoui and M. Kapałka. *Principles of Transactional Memory*. Morgan & Claypool, 2010.
8. V. Hadzilacos. A theory of reliability in database systems. *Journal of the ACM*, 35, Jan. 1988.
9. M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of ISCA'93: the 20th International Symposium on Computer Architecture*, May 1993.
10. S. Hirve, R. Palmieri, and B. Ravindran. HiperTM: High Performance, Fault-Tolerant Transactional Memory. In *Proceedings of ICDCN'14: the 15th International Conference on Distributed Computing and Networking*, Jan. 2014.

11. T. Kobus, M. Kokociński, and P. T. Wojciechowski. Hybrid replication: State-machine-based and deferred-update replication schemes combined. In *Proceedings of ICDCS'13: the 33rd International Conference on Distributed Computing Systems*, July 2013.
12. G. Korland, N. Shavit, and P. Felber. Deuce: Noninvasive software transactional memory in java. In *Proceedings of MULTIPROG '10 : 3rd Workshop on Programmability Issues for Multi-Core Computers*, Jan. 2010.
13. C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. C. Kirkham, and I. Watson. DiSTM: A Software Transactional Memory Framework for Clusters. In *Proceedings of ICPP'08: the 37th IEEE International Conference on Parallel Processing*, Sept. 2008.
14. A. Matveev and N. Shavit. Towards a Fully Pessimistic STM Model. In *Proceedings of TRANSACT '12: the 7th ACM SIGPLAN Workshop on Transactional Computing*, number 7437 in Lecture Notes in Computer Science, pages 192–206, Aug. 2012.
15. C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of IISWC'08: the IEEE International Sympozium on Workload Characterization*, Sept. 2008.
16. C. H. Papadimitrou. The Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631–653, 1979.
17. D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of OSDI '10: 9th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2010.
18. D. Perelman, R. Fan, and I. Keidar. On Maintaining Multiple Versions in STM. In *Proceedings of PODC'10: the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, July 2010.
19. M. M. Saad and B. Ravindran. HyFlow: A High Performance Distributed Transactional Memory Framework. In *Proceedings of HPDC '11: the 20th International Symposium on High Performance Distributed Computing*, June 2011.
20. N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of PODC'95: the 14th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Aug. 1995.
21. K. Siek and P. T. Wojciechowski. A Formal Design of a Tool for Static Analysis of Upper Bounds on Object Calls in Java. In *Proceedings of FMICS'12: the 17th International Workshop on Formal Methods for Industrial Critical Systems*, number 7437 in Lecture Notes in Computer Science, pages 192–206, Aug. 2012.
22. K. Siek and P. T. Wojciechowski. Brief announcement: Towards a Fully-Articulated Pessimistic Distributed Transactional Memory. In *Proceedings of SPAA'13: the 25th ACM Symposium on Parallelism in Algorithms and Architectures*, July 2013.
23. E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java Application Partitioning. In *Proceedings of ECOOP'02: the 16th European Conference on Object-Oriented Programming*, number 2374 in Lecture Notes in Computer Science, pages 178–204, June 2012.
24. A. Turcu, B. Ravindran, and R. Palmieri. HyFlow2: A High Performance Distributed Transactional Memory Framework in Scala. In *Proceedings of PPPJ'13: the 10th International Conference on Principles and Practices of Programming on JAVA platform: virtual machines, languages, and tools*, Sept. 2013.
25. R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java Optimization Framework. In *Proceedings of CASCON'99: the Conference of the Centre for Advanced Studies on Collaborative Research*, Nov. 1999.
26. A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable Transactions and their Applications. In *Proceedings of SPAA'08: the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, June 2008.
27. P. T. Wojciechowski. Isolation-only Transactions by Typing and Versioning. In *Proceedings of PPDP'05: the 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, July 2005.
28. P. T. Wojciechowski. *Language Design for Atomicity, Declarative Synchronization, and Dynamic Update in Communicating Systems*. Poznań University of Technology Press, 2007.
29. P. T. Wojciechowski, O. Rütti, and A. Schiper. SAMOA: A framework for a synchronisation-augmented microprotocol approach. In *Proceedings of IPDPS '04: the 18th IEEE International Parallel and Distributed Processing Symposium*, Apr. 2004.