Irrevocable Transactions and their Applications

Adam Welc
adam.welc@intel.com

Bratin Saha
bratin.saha@intel.com

Ali-Reza Adl-Tabatabai
ali-reza.adl-tabatabai@intel.com

Programming Systems Lab
Intel Corporation
Santa Clara, CA 95054

ABSTRACT

Transactional memory (TM) provides a safer, more modular, and more scalable alternative to traditional lock-based synchronization. Implementing high performance TM systems has recently been an active area of research. However, current TM systems provide limited, if any, support for transactions executing irrevocable actions, such as I/O and system calls, whose side effects cannot in general be rolled back. This severely limits the ability of these systems to run commercial workloads.

This paper describes the design of a transactional memory system that allows irrevocable actions to be executed inside of transactions. While one transaction is executing an irrevocable action, other transactions can still execute and commit concurrently. We use a novel mechanism called *singleowner read locks* to implement irrevocable actions inside transactions that maximizes concurrency and avoids overhead when the mechanism is not used. We also show how irrevocable transactions can be leveraged for contention management to handle actions whose effects may be expensive to roll back. Finally, we present a thorough performance evaluation of the irrevocability mechanism for the different usage models.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*; D.3.4 [**Programming Languages**]: Processors—*Run-time environments*

General Terms

Algorithms, Design, Experimentation, Languages, Performance

Keywords

concurrent programming, software transactional memory, virtual machines, performance

SPAA'08, June 14–16, 2008, Munich, Germany.

1. INTRODUCTION

Language constructs for transactional memory (TM) programming have recently gained popularity in the research community as a safer, more modular, and more scalable alternative to traditional lock-based synchronization [14, 7, 8, 5, 6]. Transactions allow programmers to compose software modules in a way that retains scalability and avoids deadlocks. Researchers have made significant progress recently on improving the performance and functionality of TM systems. Yet current systems still have restrictions that limit their practical application as a general-purpose programming tool. One of the major restrictions is the lack of support for executing *irrevocable actions* whose effects cannot in general be rolled back, such as I/O and system calls, inside transactions.

Virtually all current TM systems support *revocation transparency*: they allow the runtime system to automatically revoke any transaction (by undoing effects of its actions) on a conflict with another transaction. However, revocation transparency limits the application of transactions as it precludes execution of irrevocable actions, and may provide brittle performance for large, long-running transactions.

This paper describes the design and implementation of a TM system for Java that enables *irrevocable transactions* allowed to perform actions whose side effects either cannot be rolled back or are expensive to roll back. We relax revocation transparency by allowing a transaction to transition to an *irrevocable state* in which it will no longer roll back as a result of an external action performed by a different transaction. As a result, once a transaction transitions to an irrevocable state, the system will guarantee that its subsequent actions (including, for example, I/O and system calls) will never be revoked and that its commit operation will succeed.

We identify and concentrate on two main applications of irrevocability. The first application enables transactions to execute actions whose effects can be neither automatically rolled back nor compensated. Providing support for irrevocability is essential because it makes transactions practical in real-world programs that can perform general I/O operations inside of transactions. The second application leverages irrevocable transactions for contention management. Certain common programming scenarios, such as resizing of a concurrent data structure, can lead to situations where one transaction (i.e. , the transaction resizing the data structure) must ultimately succeed, but has a high chance of conflicting with other transactions. We show that allowing such transactions to operate in an irrevocable mode can improve overall application performance.

To summarize, we make several novel contributions in this paper:

- We introduce a lightweight mechanism called *single-owner read locks* that supports different applications of irrevocabil-

ity. Its design and implementation maximizes concurrency and at the same time avoids overhead when irrevocability is not used. (Section 3)

- We demonstrate how irrevocable transactions can be used to automatically handle execution of actions whose side effects cannot in general be rolled back, such as certain types of I/O and system calls. (Section 4)

- We show that irrevocability has another application in situations where transactions exhibit pathological contention behavior. For example, concurrent data structures, such as hash-tables, are likely to be frequently resized while they are used. This may lead to severe contention problems since the resizing transaction, requiring a long time to finish, can keep getting revoked as a result of conflicts. We introduce a new language construct that allows programmers to use irrevocability to avoid pathological contention behavior in such cases (Section 5)

- We describe a mechanism handling interactions between irrevocability and aborts that may be explicitly triggered by the programmer. (Section 6)

- We present a thorough performance evaluation of our irrevocability mechanism. Our implementation is set in a context of a high-performance, scalable STM system [1]. Our results demonstrate that our irrevocability mechanism can both (1) improve the performance of large, long-running transactions (up to over 8x) and (2) enable general-purpose I/O and system calls inside transactions while maximizing concurrency. (Section 7)

2. OVERVIEW

The independently developed TM systems often provide different language constructs to facilitate access to transactional memory by concurrently executing threads. However, recently one of the constructs, called `atomic`, has emerged as one of the most popular solutions and has been supported by several existing TM systems [1, 7, 6, 9].

The `atomic` construct specifies a block of operations executed by a thread *atomically* (no partial effects of the atomic block's execution become visible) and in *isolation* from other threads:

```
atomic {
...
}
```

The exact specification of these properties depends on the underlying transaction model (e.g. , weak atomicity vs. strong atomicity [3, 15]) and is orthogonal to the irrevocability considerations.

The implementation of the `atomic` construct involves execution of all the operations of the atomic block in a transactional context. The atomic blocks can be nested - their semantics then typically adhere to the closed nested transactions model [11]. Since in the majority of transaction models [4] it is assumed that transactions can undergo multiple revocations until they successfully commit, most TM systems also assume that all operations executed within an atomic block are *revocable*, that is, their side-effects can be automatically rolled back.

This restriction seems to be one of the major obstacles preventing wide adoption of transactions in real-world programs. Irrevocability is our proposal on how to lift this restriction. Generally speaking, we would like to introduce a new construct, `irrevocable`. This construct, when used inside of an atomic block, would guarantee that the execution of the subsequent actions would never cause the atomic block to be revoked. For example, in the figure below, we would like to ensure that no action following invocation of the method used to print the message to the screen would cause revocation of the atomic block which would in turn guarantee that the message will be printed only once:

```
atomic {
...
 irrevocable;
 System.out.println("HelloWorld");
...
}
```

In Section 4 we discuss how the runtime system can automatically trigger irrevocability to transparently handle I/O and system calls, without requiring any explicit constructs to be used.

The ability to avoid revocations and guarantee successful completion of the atomic block "in one step" can also be extremely useful for contention management purposes as it can improve throughput of long-running transactions that could otherwise be potentially revoked very frequently. While the general behavior of the basic `irrevocable` construct described here remains the same regardless of its potential use, its exact syntax and semantics can be tailored to better suit specific applications, such as contention management described in Section 5.

3. DESIGN AND IMPLEMENTATION

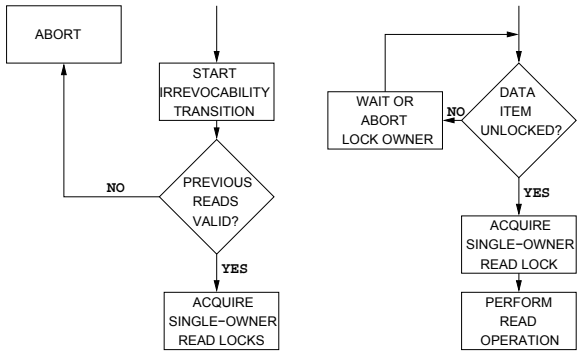
Our system uses a single foundational mechanism for supporting irrevocable transactions and maps different usage models to the underlying implementation of the basic `irrevocable` construct. Our work is set in the context of a strongly atomic [15] STM system that implements optimistic read concurrency using version numbers and pessimistic write concurrency using exclusive write locks. In STM systems using optimistic read concurrency version numbers are used to validate transactional read operations at commit time. A detailed description of this type of STM has been presented in [1, 7] and of its incarnation supporting strong atomicity in [15]. The choice of an STM has influenced the design of our irrevocability mechanism, but our mechanism can be easily adapted to other types of STMs.

3.1 Design

A trivial solution to making a transaction irrevocable is to suspend execution of all other transactions in the system to prevent them from affecting the state of the irrevocable one [1]; for example, by requiring acquisition of a single global lock. While this solution works, it greatly reduces the achievable concurrency and is unlikely to scale to highly parallel machines. One of our design goals has been to allow concurrent execution of revocable transactions with an irrevocable transaction to enable maximum throughput. Furthermore, in systems supporting strong atomicity [15], even non-transactional accesses may trigger transaction revocations. In such systems execution of all threads, not only the transactional ones, would have to be suspended before making a given transaction irrevocable.

Another solution would be to require an irrevocable transaction to always acquire an exclusive lock on each data access, regardless of whether the data item it tries to access is meant to be read or written. Moreover, in the case of a conflict, the contention manager

[1]Execution of some of these transactions may have to be revoked prior to suspension in case they hold resources, such as locks, required by the irrevocable transaction to complete its execution.



(a) transition phase (b) maintenance phase

Figure 1: Diagrams illustrating the SORL protocol phases.

would have to ensure that the execution of a revocable transaction gets revoked. However, this technique still disallows certain interleavings of operations – for example, a revocable transaction would be blocked even if it only wanted to read an item that an irrevocable transaction has also only read (i.e. , it disallows read sharing with the irrevocable transaction).

We therefore propose a hybrid protocol called *single-owner read locks* (*SORL*). Before becoming irrevocable, transactions try to acquire a token. If another transaction tries to become irrevocable while one irrevocable transaction is executing, it waits for the token to be released. If multiple transactions request a transition to irrevocability, the contention manager decides the order in which the token will be granted to each individual transaction. The requirement to allow only one active irrevocable transaction at any given time is difficult to avoid in a general case. Consider a system that allows more than one irrevocable transaction to execute concurrently. Assume that one of the irrevocable transactions requests access to a resource already held by another irrevocable transaction and vice versa. This leads to a classical deadlock situation which can only be resolved by revoking execution of one of the transaction. In this case, however, deadlock cannot be resolved since both transactions are irrevocable.

Since we allow only one active irrevocable transaction at any given time, it is only this single transaction whose revocation needs to be prevented. As a result, it is also the only one that needs to acquire and maintain read locks and the implementation of the read lock protocol can be greatly simplified. All the revocable transactions execute their reads optimistically and are allowed to read the same data items as the irrevocable one. On a write, an irrevocable transaction still acquires an exclusive lock, either directly or by upgrading a read lock. However, a revocable transaction is not allowed to write any data item that has been read-locked by the irrevocable transaction.

The SORL protocol has been designed to allow transition of a regular transaction into an irrevocable state "on-the-fly" during its execution. It consists of two phases:

1. **Irrevocability transition.** This phase is executed when a regular transaction decides to make a transition to an irrevocable state. If this phase is successful we say that the transaction successfully transitioned to an irrevocable state (became irrevocable). At the beginning of this phase the transitioning transaction first acquires the irrevocability token, then it validates its read set; that is, determines if all its read operations are still valid [2] and, if they are, acquires single-owner read locks for all data items it has read – the transition phase is successful. In case some read operations are invalid, the transition fails and the transaction releases the token, rolls back, and re-executes. In order to guarantee correctness of the transition phase, validation of a read operation and acquisition of a single-owner read lock for the read-accessed location is performed atomically – acquisition of a read lock succeeds only if transactional meta-data associated with a given data item, as described in Section 3.2, does not change since the time the read has been validated. The concept underlying the algorithm is depicted in Figure 1(a).

2. **Irrevocability maintenance.** Once a transaction becomes irrevocable, it acquires locks on accessing all data items: traditional write locks on writes and single-owner read locks on reads. The contention manager resolves all conflicts in favor of the irrevocable transaction, which may ultimately result in a revocation of any conflicting revocable transaction. This is illustrated in Figure 1(b).

3.2 Implementation

Before diving into the implementation details, we will briefly give an overview of the relevant features of McRT-STM – the TM system used for this work. McRT-STM consists of three main components: the Java virtual machine (ORP), the JIT compiler (StarJIT) and the core runtime system (McRT) providing essential transactional support. We intentionally omit the description of strong atomicity support in McRT-STM since the implementation of our irrevocability mechanism is independent on the isolation model that the base STM system supports. A more comprehensive description can be found in [1] and [15].

In McRT-STM every access to a shared data item is mediated using a *transaction record*. In case of objects, a transaction record is embedded in the object header, and in case of static variables it is embedded in the header of an object representing a class that declares this static variable (access to all static variables of a given class is mediated through the same transaction record). In case a given data item is unlocked (has not yet been updated), its transaction record contains a version number. In case a given data item is write-locked (has already been updated), its transaction record contains a *transaction descriptor pointer* pointing to a data structure containing information about the lock owner (*transaction descriptor*). These two cases can be distinguished by checking a designated low-order *lock state bit*. When a transaction record contains a transaction descriptor pointer this bit is always unset since pointers are aligned. In case a transaction record contains a version number this bit is always set because of the way version numbers are generated. Implementation of the transactional concurrency control mechanism in the original McRT-STM was carefully tuned [1, 15]. Consequently, one of our main goals when implementing the SORL protocol was to retain all the benefits of the original implementation and avoid incurring additional overheads when no irrevocable transaction is present in the system.

3.2.1 SORL encoding

In the original McRT-STM protocol, a data item could only be in one of two states: write-locked or unlocked. This information

[2] As mentioned before, the STM system used for the purpose of this work uses pessimistic writes. As a result write operations are automatically valid.

owner bit	lock state bit	description
0	0	write-locked
1	1	unlocked
0	1	unlocked
1	0	read-locked

Figure 2: SORL bit encoding.

could be encoded in the value stored in a transaction record using only one bit – the lock state bit. In the SORL protocol, however, we introduce an additional state: read-locked by an irrevocable transaction. We must therefore reserve an additional low *owner bit* to be able to encode all three states. The final result is the encoding presented in Figure 2 [3]. The lock state bit is encoded using the lowest bit of the value stored in the transaction record, while the owner bit is encoded using the second-lowest bit of this value. The remaining bits of the value stored in the transaction record contain either a transaction descriptor pointer if the data item is write-locked, or a version number in all the remaining cases. The two low-order bits are the only piece of information that need to be associated with a read-locked data item to facilitate all read lock operations – acquisition, ownership test and release – since only a single transaction operates on read locks at any time.

3.2.2 Read and Write Barriers

The implementation of transactional write barriers remains the same as in the original McRT-STM system – before being allowed to update a data item a transaction must wait for the data item to become unlocked. This can be checked, as before, by inspecting only the lock state bit.

The implementation of transactional read barriers in the original McRT-STM system has been divided into two parts: highly tuned JIT-inlined "fast-path" and the less performance critical "slow-path" implemented as a method call executed when experiencing data access contention. The code for the fast path remains the same and, in fact, behaves the same as in the original McRT-STM system if no irrevocable transactions are present, preserving all the original performance-related properties. However, in case when either an irrevocable transaction needs to acquire a single-owner read lock or a regular revocable transaction encounters a read-locked data item, the execution will fall to the slow path to acquire the read lock or check the owner bit to verify that the item is "readable", respectively.

3.2.3 Optimized logging

Irrevocability enables optimization of certain logging operations. Every regular transaction in McRT-STM uses three types of logs: a *read set* to record version numbers on reads, a *write set* to record the set of write locks acquired on writes, and an *undo log* to record original values of modified data items to support the undo operation. Since an irrevocable transaction never aborts, it does not need to maintain the undo log – before performing a data modification, a transaction checks whether it is irrevocable and elides logging of undo-related information. Additionally, an irrevocable transaction can avoid storing duplicate entries into the read set (for reads from the same location), effectively providing a perfect read set filtering mechanism for irrevocable transactions.

[3]In order to improve clarity of the presentation, the encoding presented in this paper is a subset of the one actually used in McRT-STM. The bit encoding used in McRT-STM supports other mechanisms, such as strong atomicity, whose detailed description is beyond the scope of this paper. However, both encodings have identical functional and performance-related implications.

4. IRREVOCABILITY FOR SYSTEM CALLS

Allowing arbitrary system calls to be executed inside of transactions is an obvious application of irrevocability. In Java, system calls are accessed via native methods and, as a result, modifications to the native methods subsystem were required to transparently handle system calls.

Native methods are presented to the runtime system in the form of precompiled binaries. The runtime system is then responsible for generation of the "bridge" code to allow execution of such methods from the Java code. We augmented the runtime system so that the "bridge" code uses the basic irrevocability construct described in Section 2 to perform an irrevocability transition before the native call is executed inside of a transaction. If the transaction succeeds in becoming irrevocable, it proceeds with the native method call. Initially, we augmented the "bridge" code uniformly for all native methods that were executed in the system (both internal and user-defined). However, our initial approach needed to be refined as described in Section 7.

The execution of native methods inside of a transaction is potentially unsafe for two reasons. The first reason is that the native methods may be used for actions such as I/O, and it would be incorrect to re-execute such actions. Our irrevocability mechanism addresses this since an irrevocable transaction is guaranteed to succeed. The second reason is that the native methods may not have been instrumented to allow a transaction control over arbitrary shared memory accesses (except for accesses to the Java heap, which go through JNI wrappers), and hence the memory operations from the native method may not be isolated (and atomic) with respect to memory operations from another native method executing inside a different transaction. Our implementation also takes care of this problem since at most one transaction can be executing a native method at any given time (only one irrevocable transactions can be present in the system).

5. IRREVOCABILITY FOR CONTENTION MANAGEMENT

In addition to using irrevocability to handle system calls, we would also like to use the same mechanism to help manage contention in situations when some transactions are likely to be revoked frequently as a result of conflicts. For example, consider a program using a transactional hash-table where insertions to the hash-table may cause it to be rehashed. The rehashing transaction has to access all elements in the hash-table and will likely encounter a lot of contention with other transactions accessing this hash-table at the same time. One solution that can be used to improve overall application throughput in such cases is to to trigger irrevocability automatically based on some heuristic utilizing data gathered during the application's execution to pick the most appropriate candidate for irrevocability (we present results of our experiments with one such heuristic in Section 7). Another solution is to allow a programmer to "manually" select the rehashing transaction that should operate in the irrevocable mode. When using irrevocability for contention management a programmer may wish to opt out from triggering irrevocability and potentially follow a different execution path when certain conditions are met. In order to satisfy this requirement we introduce a `cond_irrevocable` language construct whose semantics is extended with respect to the basic `irrevocable` construct (described in Section 2) to support what we will from now on call *conditional irrevocability*.

5.1 Language integration

The syntax of the `cond_irrevocable` construct in its most general form is presented below:

```
cond_irrevocable (time) { foo(); }
```

Method `foo()` is an arbitrary method returning a `boolean` value whose execution must be valid in the same scope where the `cond_irrevocable` construct is used. It specifies a condition for triggering irrevocability – if the condition evaluates to true then the irrevocability transition phase is initiated, otherwise the execution simply proceeds in a revocable form. Since only one irrevocable transaction can be active in the system at any given time, we also allow the programmer to specify the maximum amount of time a transaction is willing to wait to become irrevocable (by analogy to the conditional synchronization constructs, such as the `wait` call in Java). After the prespecified amount of time elapses, the `IrrevocableTimeoutException` is thrown to signal this event. We chose to use new syntax for the `cond_irrevocable` construct because its semantics goes beyond a simple `if (foo()) irrevocable;` statement (with timeout) as we describe in the following section.

5.2 `Cond_irrevocable` construct translation

The runtime executes the following actions when it encounters the `cond_irrevocable` construct. The method specifying the irrevocability condition is evaluated and if its evaluation yields false then the transaction remains revocable and the execution moves to the point immediately following the `cond_irrevocable` construct. Otherwise the current transaction attempts to acquire a token and proceed to the irrevocability transition phase. Introduction of conditional irrevocability raises an interesting issue. It is possible that two transactions follow the same execution path and attempt to trigger irrevocability at the same time. For example two transactions inserting into the same data structure may simultaneously decide to resize it. One transaction will become irrevocable and resize the table. At this point, the second transaction may no longer need to resize the table and hence the irrevocability condition may no longer be true. This would cause the second transaction to abort since the condition is part of its read set. To avoid this, we evaluate the irrevocability condition inside a closed nested transaction [11]. We re-evaluate the condition if a transaction fails to be irrevocable, but abort only the inner transaction.

The `cond_irrevocable` construct is implemented using Polyglot [12], an extensible source-to-source Java compiler. The syntactic form of the `cond_irrevocable` construct is translated into the code sequence presented in Figure 3. [4] The required low level transactional primitives such as basic `irrevocable` construct, nested transaction, partial rollback, and the timeout-enabled acquisition of a token are provided by the core runtime system. The token acquisition procedure may return three possible values: `SUCCESS` indicating that the token has been acquired instantaneously, `WAITED` indicating that there was contention on the token acquisition but the token was acquired before the timeout, and `FAILURE` indicating that the token acquisition operation timed out. Transaction abort in McRT-STM, as described in [1], is signaled and propagated up the call stack by throwing a special type of exception (`TransactionException`). It is up to the method implementing inner transaction commit (`txnCommitInner()`) to decide whether to propagate this exception up the call stack and abort the entire enclosing transaction (in case the triggering irrevocability was unsuccessful) or only re-execute the inner transaction (when more than one transaction attempted to become irrevocable at the same time). If the execution of the enclosing transaction needs to be revoked, it is re-started as an irrevocable one. The token is held throughout the execution of the irrevocable transaction and released upon its commit.

[4] In order to improve clarity of the presentation the code sequence is slightly simplified and it assumes irrevocability is always triggered in a context of an enclosing outer transaction.

```
while(true) {
  txnStartInner(); // start a transaction
  try {
    boolean conditionEvaluationResult = foo();
    if (conditionEvaluationResult) {
      LockOperationResult lockResult =
        acq_token(time);
      if (lockResult == FAILURE)
        throw new IrrevocableTimeoutException();
      else if (lockResult == WAITED)
        throw new TransactionException();
      else {
        // may throw TransactionException()
        irrevocable;
        break;
}  }  }
  finally {
    if (!txnCommitInner()) continue;
} }
```

Figure 3: Translation of the `cond_irrevocable (time) foo();` construct.

6. EXPLICIT REVOCATIONS

Some TM systems [1, 6] provide additional transactional constructs, such as `atomic-orelse` and `retry`, that may allow transaction revocations to be explicitly triggered. In the course of our current work we came to an unsurprising conclusion that such constructs do not easily compose with uses of the `irrevocable` construct. In order to prevent incompatible concurrent uses of irrevocability and explicit revocations triggered by the programmer, we designed and implemented a mechanism extending method signatures to propagate information required to statically detect such incompatible usage cases while still allowing the compatible ones. We now briefly describe the `retry` and `atomic-orelse` constructs. A more detailed description of their behavior and applications can be found in [6] and the discussion of their adaptation to Java can be found in [1]. We then present an overview of our solution based on a motivating example, followed by the detailed description of the mechanism.

6.1 `Retry` construct

The `retry` construct can only be used inside of an atomic block (potentially nested) and is typically used as part of a conditional. The use of this construct causes generation of a *retry event* that can be propagated up the call stack until it reaches the level of the outermost transaction. The outermost transaction and all its inner transactions are then aborted and re-executed.An example of how the `retry` construct can be used is presented below:

```
atomic {
...
 if (!condition) retry; // revoke outermost
...
}
```

In this example, a thread executing the atomic block waits for the value of the static boolean variable `condition` (assumed to initially be set to `false`) to be set to `true` by another thread.

Clearly, the semantics of the `retry` construct is incompatible with the semantics of the `irrevocable` construct, when both constructs are used on the same execution path. Consider a modified version of the example above that places the `irrevocable` construct immediately followed by a print statement between the beginning of the atomic block and the conditional containing the `retry` construct. It is unclear what would be a correct behavior in case the current transaction transitioned to the irrevocable state to protect the print statement from being re-executed (using the `irrevocable` construct) and then was asked to abort (using the `retry` construct).

6.2 `Atomic-orelse` construct

The situation is additionally complicated in presence of another construct, the `atomic-orelse`, which is used to compose two or more alternative transactions [1, 6]. This construct also has to be used inside of a potentially nested atomic block. Let us consider a case when only two alternatives are being composed. The execution starts with the first alternative being executed as a nested transaction. If it completes successfully, the execution of the whole `atomic-orelse` construct is successful as well. If during the execution of the first alternative the `retry` construct is used, the retry event is generated but its propagation stops at the level of the nested transaction, whose execution then gets revoked. Then the second alternative is executed as a nested transaction. However, the use of the `retry` construct within the second alternative causes the retry event to be propagated to the level of the outermost transaction enclosing the `atomic-orelse` construct. Then the execution of the entire outermost transaction gets revoked and the whole process of selecting an alternative retried.

An example of how the `atomic-orelse` construct can be used is presented below, where a thread executing the code fragment chooses one of the two alternatives based on one of the conditions (assumed to initially be set to `false`) being set to `true` by other threads.

```
atomic {
...
 atomic {
 ...
 if (!condition1)
  retry; // revoke nested only
  ...
 }
 orelse {
  ...
  if (!condition2)
   retry; // revoke outermost
   ...
 }
...
}
```

It is straightforward to generalize the behavior of this construct to more than two alternatives – if the second alternative fails the remaining alternatives keep getting executed until one of them succeeds or the last one retries in which case the retry event gets propagated to the level of the outermost enclosing transaction.

6.3 Overview

Our system uses a combination of compile-time analysis and dynamic *irrevocability suspension* technique to make irrevocability coexist with transaction retries. We statically rule out a possibility of a transaction trying to become irrevocable and attempting a retry within the same scope. However, we allow a limited use of the `retry` construct in an inner transaction nested in some outer irrevocable transaction provided that the retry event does not propagate to the outer transaction. This way, we can safely execute irrevocable actions in the outer transactions and allow the inner transaction to retry as long as the inner transaction itself does not trigger irrevocability. An example of such situation is presented below, where it is safe to retry the inner transaction since the use of the `retry` construct will never cause revocation of the print statement – control will be passed to the `orelse` clause which does not retry.

```
atomic {
...
 irrevocable;
 System.out.println("HelloWorld");
 atomic {
  ... retry; ...
 }
 orelse {
  ... // does not retry
 }
...
}
```

We will first present a set of additional examples to illustrate rules that need to be followed when programming with retry and irrevocability and then present the mechanisms used to guarantee that the rules are indeed being obeyed. We would like to emphasize that these mechanisms are intended to be used when irrevocability is applied to handling I/O and system calls, in which case their presence is essential, but they can also be adapted to help programmers use irrevocability for contention management (for example, by having compile-time analysis generate warnings instead of errors).

6.4 Programming with retry and irrevocability

We define the following set of rules for the `retry` construct and the `atomic-orelse` construct if they are encountered within an irrevocable transaction:

1. The use of the `retry` construct enclosed only by one or more (nested) simple atomic regions (none of which is part of the `atomic-orelse` construct) should be forbidden. In such a case we do not know up front how far the retry event will be propagated. It is therefore possible for the retry event to propagate even beyond the point when irrevocability was triggered.

2. The use of the `retry` construct within the Nth alternative of the `atomic-else` construct consisting of N alternatives should be forbidden for the same reason as above.

3. The use of the `retry` construct within any of the first N-1 alternatives of the `atomic-else` construct consisting of N alternatives should be allowed provided that irrevocability is not triggered within the scope of any of the N-1 alternatives. We allow this behavior since the retry event will not get propagated to the outer, potentially irrevocable, transaction.

In Figure 4 we illustrate how the rules described above work by presenting sample code sequences (using only transactional constructs for brevity) that should be allowed or disallowed. We use the basic `irrevocable` construct since the rules are meant to be followed mostly in case irrevocability is used to handle I/O and system calls when the same type of construct is used (as described

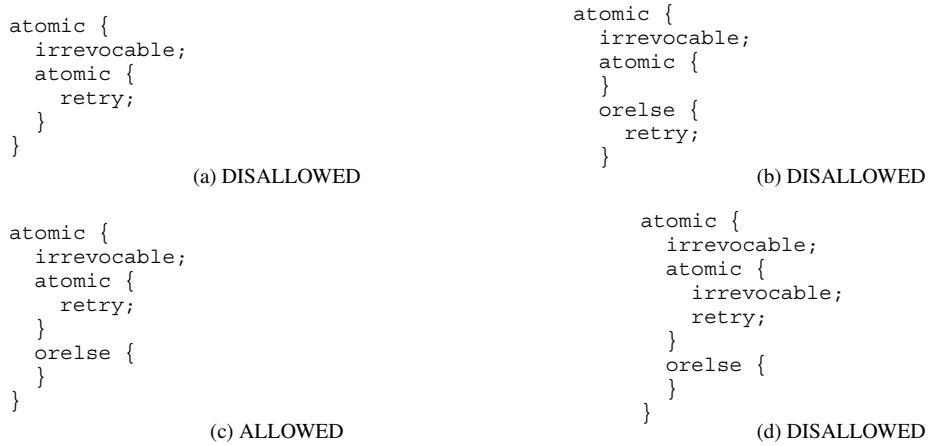


Figure 4: Usage of `retry` and `atomic-orelse` constructs in presence of irrevocability.

in Section 5). The code sequence in Figure 4(a) should be disallowed by rule number 1, otherwise the retry event will propagate beyond the point where the transaction became irrevocable. The code sequence in Figure 4(b) should be disallowed for the same reason by rule number 2. The code sequence in Figure 4(c) should be allowed by rule number 3 (the retry event is confined within `atomic-orelse` and will never propagate beyond point when irrevocability was triggered), while the code sequence in Figure 4(d) should be disallowed by the same rule (the retry event is confined but irrevocability is triggered again in the scope of confinement).

6.5 Compile-time analysis

Ideally a compile-time analysis would detect and disallow all the sequences that are forbidden according to the rules described previously, and only those sequences. However, such an analysis is in general not possible since it would require the ability to precisely determine all the control flow paths of the application. For example, presence of the `retry` construct in the conditional branch that is never taken is always correct regardless of whether it is confined within the `atomic-orelse` construct or not. In our system we implemented a flow-insensitive compile-time analysis that rules out incorrect programs, but may also conservatively rule out some additional programs.

We extend a signature of Java methods to be potentially extended with one of the following qualifiers:

- `norevoke` – a method whose signature is extended with the `norevoke` qualifier contains a direct (within the body of the method) or indirect (through a chain of method calls) use of the `irrevocable` construct that can potentially make the enclosing outer transaction irrevocable
- `mayretry` – a method whose signature is extended with the `mayretry` qualifier contains a direct or indirect use of the `retry` construct that can potentially cause propagation of the retry event to the level of the enclosing outer transaction

For the purpose of the compile-time analysis, we treat the `retry` construct and the `irrevocable` construct as methods whose signatures automatically contain appropriate qualifiers.

The following rules concerning the qualifiers must be enforced by the modified Java compiler:

1. The signature of every method containing a direct use of the `irrevocable` construct must contain the `norevoke` qualifier.
2. The signature of every method containing a direct use of the `retry` construct must contain the `mayretry` qualifier, unless all uses of the `retry` construct are confined to one of first the N-1 alternatives of the `atomic-orelse` construct consisting of N alternatives.
3. For every method of a given class or interface whose signature contains a qualifier, signatures of all methods from the super-classes or super-interfaces that this method either must contain the same qualifier.
4. Similarly to rule number 1, the signature of every method containing a call to another method whose signature contains the `norevoke` qualifier must also contain the `norevoke` qualifier.
5. Similarly to rule number 2 (and under the same condition), the signature of every method containing a call to another method whose signature contains the `mayretry` qualifier must also contain the `mayretry` qualifier.

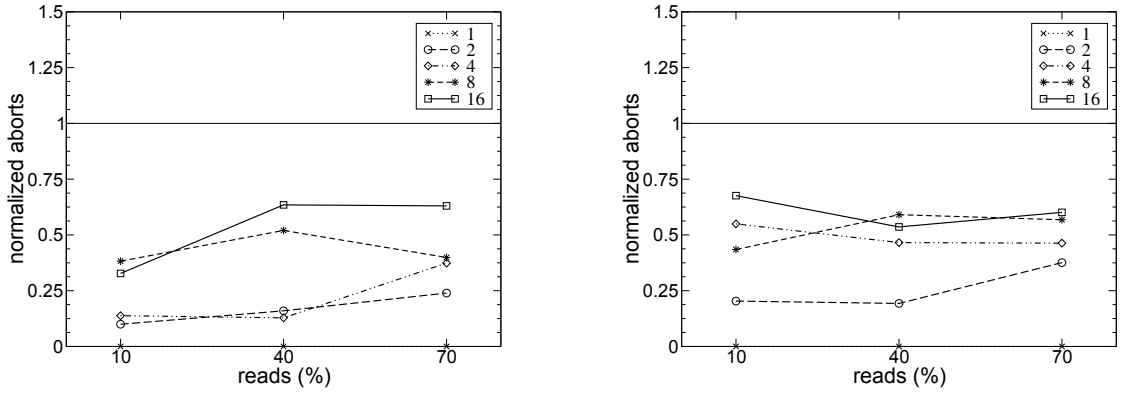
As a result, the qualifiers become part of the explicit contract that governs interactions between different parts of the same application (e.g. , some library code and user-level code using this library).

Despite Java's dynamic method dispatch mechanism, this information required to enforce rule 4 and rule 5 is available at compile time thanks to rule number 3.

Once the placement of the qualifiers has been verified by the compiler according to the rules listed above, only a trivial check is required to verify if the retry event has a chance to propagate to the scope of an irrevocable transaction – it will be the case if two methods whose signatures contain different types of qualifiers could be executed in the same scope.

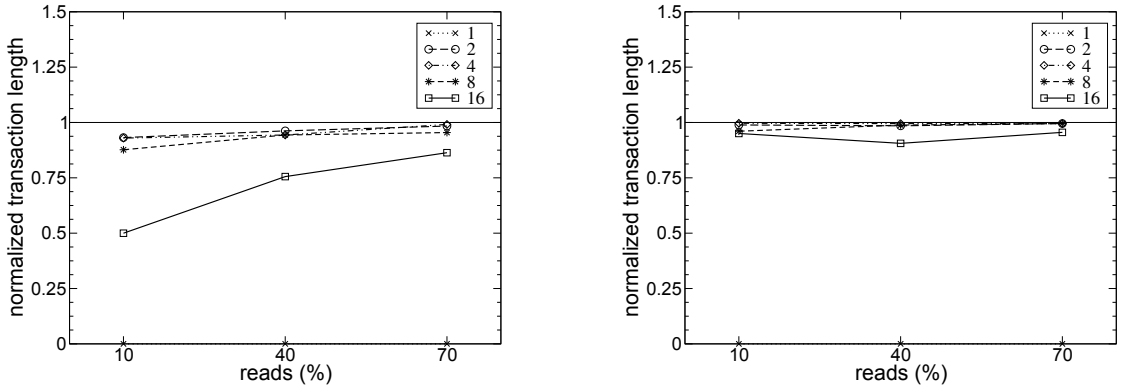
6.5.1 Implementation

When implementing our compile-time analysis, we tried to avoid further (beyond introduction of the TM-related constructs) modifications to the Java language. We overload semantics of the Java's `throws` clause and define two special types of *qualifier exceptions*, never instantiated at runtime, to serve as qualifiers. An additional advantage of using the `throws` clause to propagate information about the `irrevocable` and the `retry` constructs is that methods not using these constructs explicitly but only propagating information about their use can be compiled using a standard Java compiler that does not support any transactional extensions.



(a) 200k ops (b) 20k ops

Figure 5: Number of rehashing transaction aborts normalized with respect to total number of aborts.



(a) 200k ops (b) 20k ops

Figure 6: Length of the rehashing transactions normalized with respect to length of all transactions.

We implemented our analysis in the Polyglot compiler [12] by introducing an additional compiler pass, modeled after the standard exception checker, but processing the qualifier exceptions separately from all other types of exceptions. The new compiler pass checks that the qualifier exceptions are propagated correctly, and that no two methods throwing different types of qualifier exceptions are present in the same scope. The only additional rule that needed to be enforced was to generate a compile-time error in case of an attempt to suppress propagation of a qualifier exception using a `try-catch` clause.

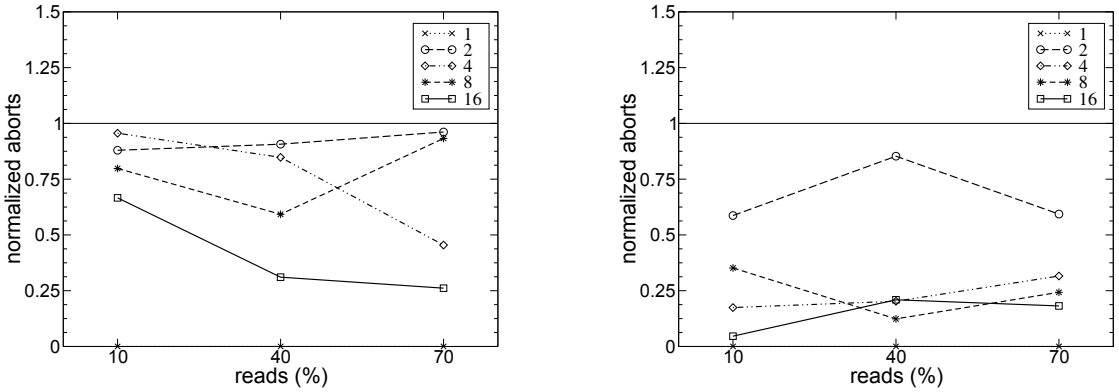
6.6 Irrevocability suspension

In order to match the behavior of the `irrevocable` and `retry` constructs described earlier in this section we had to modify the implementation of our irrevocability mechanism described in Section 3. Previously we assumed that once a transaction became irrevocable, no part of it would need to be (or even could be) rolled back. This may no longer be true since the irrevocable transaction could encounter a `retry` construct in a nested transaction.

We therefore introduce the notion of *irrevocability suspension*. An irrevocable transaction that starts one of the N-1 alternatives of the `atomic-orelse` statement consisting of N alternatives dynamically suspends its irrevocability. When irrevocability of a transaction gets suspended, the operations of this transaction can be rolled back but only as a result of the `retry` construct – other transactions still cannot affect the execution of a transaction whose irrevocability has been suspended. In a suspended state, an irrevocable transaction must always maintain an undo log and therefore cannot use optimized logging described in Section 3.2.3. It can, however, still use the read set filtering described in the same section. Note that irrevocability needs to be suspended only when a transaction starts executing an alternative of the `atomic-orelse` construct that can potentially retry. In turn, this can be detected statically. Irrevocability is resumed when the transactions goes back to the scope where retry cannot happen anymore – the runtime system tracks the depth appropriately for this.

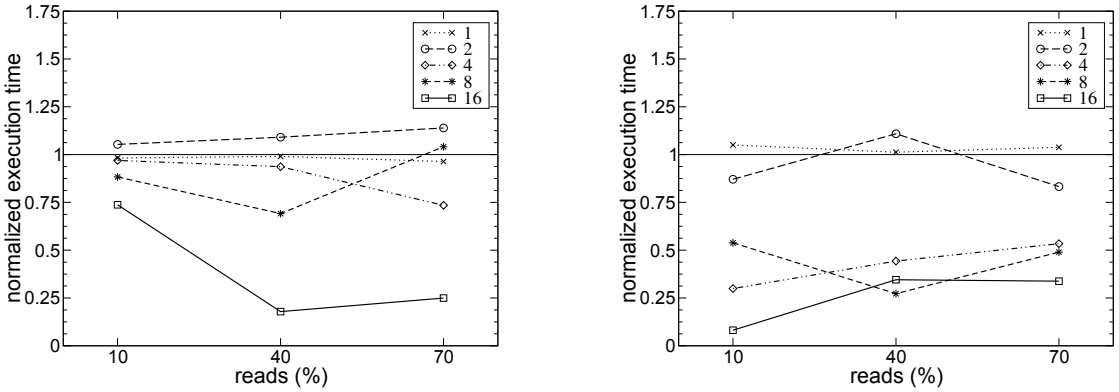
6.7 Discussion

Our choice of qualifiers described earlier in this section represents only one point in a design space. In our system, "regular" transactions can neither become irrevocable (i.e. , trigger irrevocability) nor retry (i.e. , generate a retry event) – we consider both irrevocable transactions and retrying transactions as special cases and use qualifiers to make these cases explicit to the programmer. Alternatively, one could envision a solution where it is perfectly legal for a "regular" transaction to become irrevocable. Qualifiers



(a) 200k ops (b) 20k ops

Figure 7: Number of aborts – implicit irrevocable vs. revocable (normalized).



(a) 200k ops (b) 20k ops

Figure 8: Average elapsed time – implicit irrevocable vs. revocable (normalized).

could then be used to mark as special cases transactions that can never become irrevocable (i.e. , never trigger irrevocability) and those that can potentially retry (i.e. , can generate a retry event). Obviously a different set of rules would have to be defined to provide similar guarantees to the ones provided by the solution we presented in the appendix. We defer further exploration of this design space to future work.

7. PERFORMANCE EVALUATION

We first present the results of using irrevocability for contention management. All our experiments were performed on an IBM xSeries 440 machine running Windows 2003 Server Enterprise Edition. This machine has 16 2.2GHz Intel® Xeon® processors and 16GB of shared memory arranged across 4 boards.

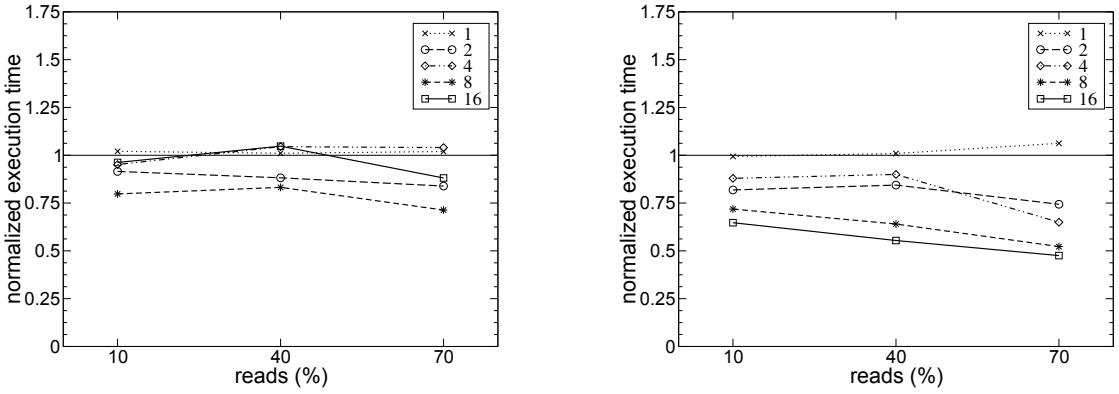
7.1 Contention management

We use irrevocability to guarantee successful and timely completion of transactions that are potentially long-running and likely to abort frequently. We use a hash-table as an example – most commercial implementations rehash the hash-table when the load factor exceeds a threshold. We investigate if irrevocability can improve performance of hash-table operations in the presence of rehashing using a transactional version of a well-known and highly efficient implementation of the hash-map data structure created by Doug Lea [10]. The transactional version of this hash-map was created by converting synchronized regions into transactional regions. The transactional version has been previously shown to perform as well as the synchronized one [1]. We therefore concentrate on comparing the performance of versions that use irrevocability (implicitly or explicitly – as described below) with a version that does not.

We run multiple configurations for both revocable and irrevocable versions of the system, varying the following parameters:

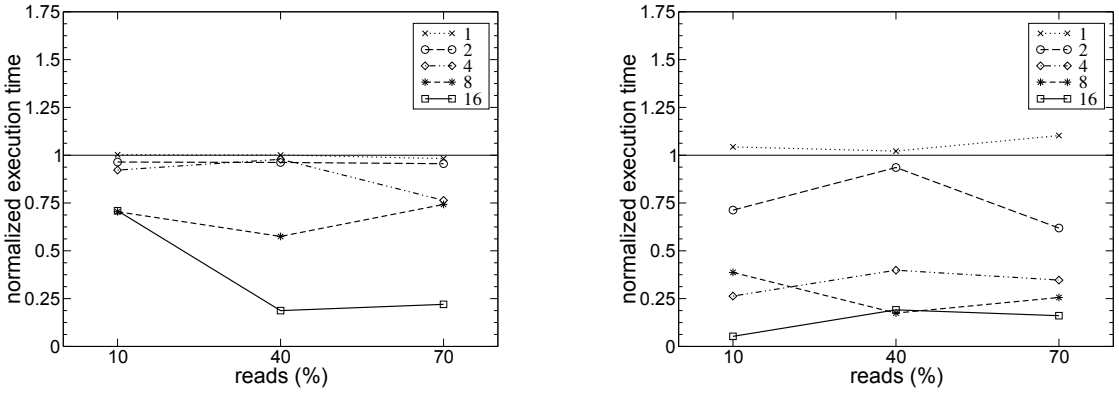
- number of threads: 1, 2, 4, 8 or 16
- total number of operations (distributed uniformly among all threads): 200k, 100k, 50k or 20k
- distribution of types of hash-map operations: 10%, 40% or 70% of lookups; the remaining N operations (where N=100%-lookup%) comprised of two-thirds insertions and one-third deletions.

We ran all configurations using the finest available granularity of conflict detection – word-level for objects and element-level for arrays. Our measurements are based on executing each configuration



(a) 200k ops (b) 20k ops

Figure 9: Average elapsed time – implicit irrevocable vs. explicit irrevocable (normalized).



(a) 200k ops (b) 20k ops

Figure 10: Average elapsed time – explicit irrevocable vs. revocable (normalized).

100 times (discarding the compilation cost). Despite using a large number of iterations, there was some jitter in our results. This is caused by re-hashings being triggered nondeterministically (by the first transaction that observes threshold being exceeded) as well as due to the configuration of the machine (it is arranged as clusters of 4 CPUs which leads to somewhat unpredictable communication cost). The general trends represented by these numbers are, however, quite clear and allow us to draw sound conclusions with respect to the overall performance of our system. In order to simplify the analysis of the results, we present and discuss only numbers for configurations executing 200k and 20k operations – the results for "intermediate" configurations, unsurprisingly, represent the middle ground between the two "boundary" configurations and do not add anything significant to the discussion.

The first experiment estimates the potential gain from applying irrevocability to hash-map rehashing. In Figure 5 we plot the number of aborts for the rehashing transactions normalized with respect to the total number of aborts by all transactions. We observe that the number of rehashing transaction aborts can amount to almost 75% of the total aborts. The next set of graphs, in Figure 6, plots the length of aborted rehashing transactions normalized with respect to length of all aborted transactions. We wanted to estimate the work wasted due to the aborts of the rehashing transaction. We use the length of the read set as a proxy for the length of a transaction. The figure shows that rehashing transactions dominate the total abort cost – for some configurations nearly all the aborts can be attributed to the rehashing transaction. The analysis of both figures indicates that irrevocability would be a good candidate for improving performance if we managed to use it for reducing the number of aborts of the rehashing transaction.

It would certainly be most convenient to trigger irrevocability "under-the-hood" without requiring explicit programmer's intervention. We experiment with one simple heuristic that could be used for that purpose. We categorize all transactions into two groups:

1. transactions that abort at most once – their execution is not particularly amenable to failure (their abort is "accidentally" caused by an unfortunate interleaving of their operations with operations of other transactions)

2. transactions that abort multiple times – their execution is for some unknown reason abort-prone (for example, they may be long-running)

We then trigger irrevocability on the *second* abort of a given transaction in hope of improving performance for transactions from the second group, such as rehashing transactions in the case of the

hash-map benchmark. Our categorization is obviously simplified – even transactions that are not particularly amenable to failure may be aborted several times, which could result in multiple transactions competing to enter irrevocable mode. In order to avoid serialization of these transactions we make acquisition of irrevocability token (described in Section 3.1) non-blocking – if token is already acquired, transaction proceeds in non-irrevocable mode.

Our goal when designing this experiment was to verify if irrevocability can be successfully used to automatically improve average performance of the transactional workloads. The analysis of Figure 7 and Figure 8 provides an evidence that this is indeed the case – even when using a rather simple heuristic we observe significant reduction in the number of aborts and up to over 4x speedup in the majority of multi-threaded runs. Because of its simplicity, our heuristic does lead to some performance degradation in some cases. At the same time, we by no means claim to have exhausted the space of solutions that can be used to decide if (and when) irrevocability should be triggered. One possible solution to further improving performance is to use more sophisticated heuristics [5], which is an interesting research area on its own that, however, goes beyond the scope of this paper. Another option is to rely on a programmer to utilize application-specific knowledge and explicitly designate transactions that should run in the irrevocable mode – our third set of experiments verifies whether utilization of such knowledge can indeed be beneficial.

In the third set of experiments we execute the same hash-table operations, but every transaction that is about to execute a rehashing routine is explicitly marked as irrevocable using the conditional irrevocability construct described in Section 5 (with infinite timeout). The evaluation of the irrevocability condition ensures that no two transactions will attempt rehashing at the same time. In Figure 9 we plot the average elapsed times for configurations with rehashing transactions explicitly marked as irrevocable normalized with respect to the average elapsed time for configurations implicitly triggering irrevocability. We can observe that in almost all multi-threaded runs performance has been improved (up to 2x) even with respect to configurations whose performance was already in most cases better than that of configurations that did not use irrevocability (Figure 8). The reason single-threaded runs may suffer some penalty here is that for these configurations the cost of triggering irrevocability must be paid even for single-threaded runs, while for the others it does not. [6]. Even in the single-threaded cases, however, we observe that the overheads are relatively low. With respect to comparison between multi-threaded configurations with rehashing transactions explicitly marked as irrevocable and those not using irrevocability at all, the former deliver up to over 8x speedup and in all cases perform no worse or marginally worse than the latter (Figure 10).

7.2 Native method calls

We evaluate performance of the irrevocability mechanism used to handle native method calls using a transactional version of a well known Java benchmark emulating a 3-tier system for a wholesale company with multiple warehouses – SPECjbb2000 [17]. The transactional version of SPECjbb2000 (*txnJBB*) was created by replacing the Java synchronization constructs (monitor acquisition and release) with their transactional equivalents (transaction start and commit).

[5]For example, transaction size could be taken into consideration.

[6]Non-irrevocable configurations obviously do not trigger irrevocability at all and implicitly-irrevocable configurations do not trigger it in single-threaded cases due to lack of aborts.

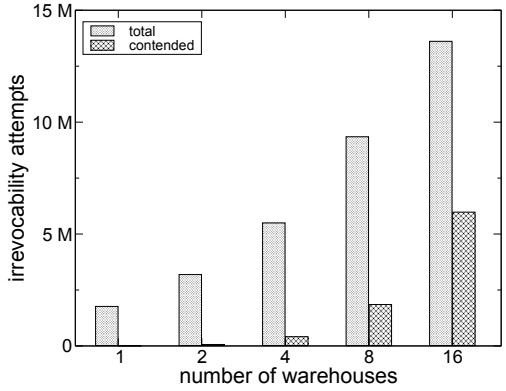


Figure 11: Number of irrevocability attempts – irrevocable vs. revocable (normalized).

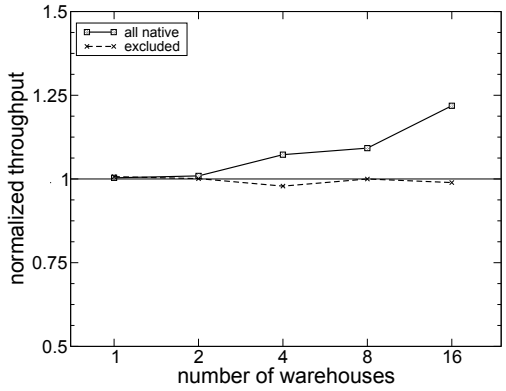


Figure 12: Average throughput – irrevocable vs. revocable (normalized).

We ran our set of experiments using txnJBB varying the number of warehouses (and correspondingly threads) from 1 to 16. We ran 10 iterations of each configuration (excluding compilation cost). We wanted to measure whether the serialization of the irrevocable transactions affected total throughput. We therefore compare the irrevocable version of the system (implicitly making the current transaction irrevocable before it calls a native method, as described in Section 4) with the revocable one. Note that the revocable version may be potentially unsafe since I/O operations and native methods inside transactions may get re-executed – however the revocable version provides an upper bound on the performance. Our initial performance (throughput) comparison indicated that uniform application of irrevocability for all native method calls does affect the overall performance, especially as the number of warehouses (and therefore threads) grows since transactions end up waiting to be irrevocable. In Figure 11 we plot both the number of total irrevocability attempts [7] (the first bar) and the number of contended irrevocability attempts (the second bar) when a transaction needed to wait for the currently irrevocable one to complete.

To remedy the problem we identified a small subset of internal native methods without any irrevocable side-effects. These include methods returning object's hash-code, methods implement-

[7]All the attempts were successful due to the low contention on data accesses and thus possibility for a conflict-induced abort.

ing numerical computations in the `java.lang.Math` class (e.g. , trigonometric functions), a method returning a string representation of a `java.lang.Double` object and a method natively creating a Java object. As indicated on Figure 12, exclusion of these few methods was sufficient to make the performance of the irrevocable version comparable to the revocable one. In Figure 12 we plot average throughput for configurations using irrevocability uniformly on all native methods (solid line) and the one excluding benign native methods from triggering irrevocability (dashed line) – both normalized with respect to the average throughput for configurations using the revocable version of the system. The former induces up to 18% overhead while the latter has virtually no overhead when compared with the revocable version.

The results in this section demonstrate that irrevocability can be used for executing actions that are difficult to undo inside transactions without paying a performance penalty.

8. RELATED WORK

In parallel with this work Spear et al. [16] introduced a concept similar to irrevocability – *inevitable transactions*. They present and evaluate several different implementations of inevitable transactions and explore their ability to handle system calls and to improve performance of transactional applications. As a result of using a library-based STM they do not explore transparent handling of I/O and system calls through automatic injection of "inevitability" triggers. In their system, they forbid explicit user aborts inside of inevitable transactions, but a mechanism that could enforce such property is not discussed.

Martin et al. [2] introduce the notion of *unrestricted transactions* in a context of a hardware transactional memory system. The concept of unrestricted transactions is similar to irrevocability. However, in order to support their mechanism, they rely on a complex hardware support that requires changing the cache coherence protocol as well as maintaining additional metadata in the memory hierarchy. Also, their solution does not allow "on-the-fly" transition of a regular transaction into the irrevocable state – a transaction that wants to become unrestricted must first be aborted. It is unclear how their proposal would be integrated in a language environment, or would be used for contention management.

Welc et al. [18] describe a solution to handling system calls in a restricted setting. In their system it is always safe for a transaction to revert to using mutual-exclusion for synchronization. As a result, they can handle system calls by reverting to using mutual-exclusion right before such calls are executed. Naturally, their solution is only applicable to systems where both types of concurrency control mechanisms (transactions and mutual-exclusion) are not only allowed to co-exist but also provide the same synchronization semantics. Such systems typically come with their own set of restrictions and have not yet managed to gain wide popularity. A similar solution, though established in a context of hardware transactional memory, has been proposed by Rossbach et al. [13]. In their system, a thread attempting to execute a critical section containing I/O operations can acquire an exclusive lock preventing all other threads, both transactional and non-transactional, from concurrently executing the same critical section.

9. CONCLUSIONS

In this paper we have discussed the design and implementation of irrevocable transactions – that is, transactions that are guaranteed to commit. We explored two applications of irrevocability, one to safely execute general-purpose system calls from inside of transactions and the other to use irrevocability for contention management. We showed that irrevocability can significantly improve performance in many common programming situations. We also showed that irrevocability can be used for handling actions such as I/O which are difficult or impossible to undo.

10. REFERENCES

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI 2006*.

[2] C. Blundell, E. C. Lewis, and M. Martin. Unrestricted transactional memory: Supporting i/o and system calls within transactions. Technical Report CIS-06-09, University of Pennsylvania, Department of Comp. and Info. Science, 2006.

[3] C. Blundell, E. C. Lewis, and M. M. K. Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), Nov 2006.

[4] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Data Management Systems. Morgan Kaufmann, 1993.

[5] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA 2003*.

[6] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP 2005*.

[7] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI 2006*.

[8] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA 2006*.

[9] B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *MSPC 2006*.

[10] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Welsey, 1999.

[11] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, Massachusetts, 1985.

[12] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: an extensible compiler framework for Java. In *CC 2005*.

[13] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, A. Bhandari, and E. Witchel. TxLinux: Using and managing transactional memory in an operating system. In *SOSP 2007*.

[14] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. A high performance software transactional memory system for a multi-core runtime. In *PPoPP 2006*.

[15] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and S. Bratin. Enforcing isolation and ordering in STM. In *PLDI 2007*.

[16] M. Spear, M. Michael, and M. Scott. Inevitability mechanisms for software transactional memory. In *TRANSACT 2008*.

[17] Standard Performance Evaluation Corporation (SPEC). SPECjbb2000 benchmarks, 2000. http://www.spec.org/jbb2000.

[18] A. Welc, A. L. Hosking, and S. Jagannathan. Transparently reconciling transactions with locking for Java synchronization. In *ECOOP 2006*.