# Atomic Boxes: Coordinated Exception Handling with Transactional Memory

Derin HARMANCI[⋆] Vincent GRAMOLI[⋆‡] Pascal FELBER[⋆]

[⋆] University of Neuchâtel, Switzerland
[‡]EPFL, Switzerland

**Abstract.** In concurrent programs raising an exception in one thread does not prevent others from operating on an inconsistent shared state. Instead, exceptions should ideally be handled in coordination by all the threads that are affected by their cause.

In this paper, we propose a Java language extension for coordinated exception handling where a named `abox` (atomic box) is used to demarcate a region of code that must execute *atomically* and in isolation. Upon an exception raised inside an `abox`, threads executing in dependent `abox`es, roll back their changes, and execute their recovery handler in coordination. We provide a dedicated compiler framework, CXH, to evaluate experimentally our atomic box construct. Our evaluation indicates that, in addition to enabling recovery, an atomic box executes a reasonably small region of code twice as fast as when using a failbox, the existing coordination alternative that has no recovery support.

**Keywords:** error recovery, concurrent programs, failure atomicity.

## 1 Introduction

Exceptions and exception handling mechanisms are effective means for redirecting the control flow of an error-prone sequential program before it executes on an inconsistent state of the system. This fact has led to extensive studies on exception handling mechanisms and their being tailored to work well with sequential programs. At the same time, a recent survey on 32 sequential applications presents the general picture on the exception handling usage by the programmers and reports that even though more than 4% of the total source code is dedicated to it, exception handling is still neglected in most of the cases: either terminating the program or ignoring the exception [1]. This result shows that sequential programs are generally developed by using exceptions as a means to terminate programs in a convenient way and inconsistencies resulting from exceptional situations are not really treated.

In concurrent programs, however, an exception raised by one thread cannot prevent other threads from accessing an inconsistent shared state because other threads may not be aware of the raised exception. Such an exception should ideally be detected by all the threads that operate on the same shared state because they can be affected by its cause. Two solutions to the problem can be considered: *(i)* the program should be brought back to a consistent state by handling the exception, or *(ii)* all the affected threads (or even the whole application) should be terminated to avoid execution on inconsistent shared states. Since there are no widespread mechanisms that allow any of these solutions, it is up to programmers to devise a solution for such cases. In other words, compared to sequential programs where treating exceptions is barely considered, for concurrent programs handling exceptions should be part of the main application design and development in order to not jeopardize the application correctness.

To illustrate how easily the above inconsistency problem can appear in ordinary concurrent programs, consider the following code in Figure 1 (inspired by a similar example in [2]). The figure presents a naive implementation of a classifier program where multiple threads concurrently evaluate nodes from the `unclassifiedNodes` list, process them, and move them to the target class using the `assignToClass` method. Note that we assume that both the `unclassifiedNodes` list and the target classes `class[N]` are shared by all threads.

```
1  Class NodeClassifier {
2    int N; // number of classes
3    List unclassifiedNodes;
4    Set class[N];
5    ...
6    public void assignToClass(int srcPos, int targetClass) {
7      synchronized(this) {
8        Node selectedNode = unclassifiedNodes.remove(srcPos);
9        selectedNode.transform();
10       class[targetClass].add(selectedNode);
11     }
12   }
13 }
```

**Fig. 1.** A concurrent code that may end up in an inconsistent state if an exception is raised while the selected node's representation is being transformed as required by the target class object in `selectedNode.transform()`.

When an exception is raised on line 9, the system reaches an inconsistent shared state if the exception is not handled: the `selectedNode` gets lost as it is neither in the `unclassifiedNodes` nor in its target class. For correct execution of the program, the exception should be handled and this should be performed before any of the other threads, unaware of the raised exception, access either the `unclassifiedNodes` list or the target class, which are inconsistent. Hence, the handling of the exception should take the existence of concurrent threads into account.

This example, albeit naive, clearly shows that exception handling becomes a first class design consideration in development of correct concurrent programs. And, needless to say, with the mainstream computer hardware becoming multi-core, concurrent programming is about to become mainstream too. This fact highlights the need for solutions that will simplify concurrent programming under exceptional situations.

Recently, Jacobs and Piessens proposed *failbox* as a mechanism to prevent the system from running in such an inconsistent state. The key idea is that, if one thread raises an exception in a failbox, any other thread is prevented from executing in the same failbox [3]. Instead of letting the system run in an inconsistent state, a failbox simply halts all concurrent threads accessing the same failbox. However, failboxes neither revert the system to a consistent state nor help the programmer recover from the error.

In this paper, we propose an `abox-recover` construct as a language extension that supports coordinated exception handling by providing `abox` and `recover` blocks (the keyword `abox` is derived from "atomic box"). Our `abox-recover` construct differs radically from the failbox extension, as it reverts the system to a consistent state upon exception to enable recovery through coordinated exception handling. Hence, `abox`es do not only propagate the exceptions to concurrent threads of the system (as failboxes do), but also allow these threads to recover from this exception in a coordinated manner.

The programmer uses a named `abox` to demarcate regions of code that should remain in a consistent state upon exception. The `abox` guarantees failure atomicity either by executing all its content or by reverting its modifications. Failure atomicity allows the programmer to handle exceptions. For example, by replacing `synchronized` with `abox` in Figure 1, the inconsistency problem can be solved: the `abox` reverts all its changes including the modification of `unclassifiedNodes`. Dependencies between `abox`es are indicated using a simple naming scheme: *dependent* `abox`es (ones that are subject to inconsistencies related to the same data) are attributed the same name. If an exception is raised inside an `abox`, all threads executing in dependent `abox`es stop their execution and rollback their changes. Then, execution continues in a `recover` block, analogous to a `catch` block, by one or all the affected threads as specified by the programmer. Typically, the `recover` block aims at correcting the condition that caused the exception and/or reconfigure the system before redirecting the control flow, restarting for example the execution of the atomic boxes. Our `abox-recover` construct therefore provides the simplicity of a `try-catch`, but for coordinated exception handling in multi-threaded applications.

**Contributions.** We propose an `abox-recover` construct as a language extension for coordinated exception handling. Our `abox` block uses *transactional memory* (TM), a concurrent programming paradigm ensuring that sequences of memory accesses, *transactions*, execute atomically [4]. As far as we know, `abox` is the first language construct that benefits from memory transactions for concurrent exception handling.

More specifically, an `abox` acts like a transaction that either *commits* (all its changes take effect in memory), or *aborts* (all its effects are rolled back). The main difference between `abox` and memory transactions lies in the way commit and abort are triggered. In TM, transactions abort only if a detected conflict prevents the transaction from being serialized with respect to concurrently executing transactions. With coordinated exception handling, an `abox` is rolled back also if an exception has been raised inside a dependent `abox`, which leads to the execution of the corresponding `recover` block.

We have implemented a compiler framework for coordinated exception handling, called CXH, that converts `abox`es into some form of transactions. Our CXH compiler framework ensures that all `abox`es execute speculatively, making sure that no exceptions are raised before applying the changes of the corresponding `abox`es in the shared memory. More precisely, CXH consists of a dedicated Java pre-compiler that converts our language extensions into annotated Java code, which is executed using a TM library thanks to an existing bytecode instrumentation agent. The CXH compiler generates code that guarantees that, if an exception is raised in an `abox`, each thread executing a dependent `abox` concurrently gets notified and rolls back the changes executed in the corresponding `abox`. Depending on the associated `recover` block, the threads perform appropriate recovery actions and restart or give up the execution of the `abox`.

We compare experimentally our `abox-recover` construct against failboxes, which only stop threads running in the same failbox without rolling back state changes. Our results indicate that `abox`es that comprise up to few hundreds memory accesses execute 2× faster than failboxes in normal executions, where no exceptions are raised, and 15× faster than failboxes to handle exceptions. We also tested extreme cases where an `abox` executes thousands of memory accesses, in which case the cumulated overhead of TM accesses may result in lower performance than long failboxes. Besides illustrating that TM is a promising paradigm for failure atomicity and strong exception safety, our evaluation indicates that the `abox` mechanism is efficient compared to similar techniques providing weaker guarantees.

**Roadmap.** Section 2 presents the background and related work. Section 3 introduces an example that is used subsequently to illustrate our language constructs. Section 4 describes the syntax and semantics of the language constructs for coordinated exception handling. Section 5 presents the implementation of coordinated exception handling and our CXH compiler framework. Section 6 compares the performance we obtained against failboxes and Section 7 concludes.

## 2 Background and Related Work

**Concurrent recovery.** Thanks to their ability to rollback and their isolation from the other parts of the program, atomic transactions have been used for concurrent handling of exceptions since the eighties [5]. Transactions by themselves have been considered useful only for *competitive concurrency* where concurrently executing threads execute separately, unaware of each other, but access common resources. This type of concurrency is the primary target of our approach.

A classical alternative to avoid inconsistencies in portions of programs generating competitive concurrency consists in encapsulating the associated code in transactions. Argus [6], Venari/ML [7] and Transactional Drago [8] map transactions to methods (for which multiple threads can be spawned) and allow an exception that cannot be resolved on a local thread to abort the transaction, passing the exception to the context where the method is called. OMTT [9] allows existing threads to join a transaction but still propagate exceptions to a context outside the transaction. In these approaches, exceptions concerning all the competing threads result in the rollback of the transaction and the propagation of the exception outside the transaction. In contrast, our approach allows threads to (cooperatively) handle such exceptions, instead of directly propagating the exception outside of the transaction scope.

The secondary target of our approach is *cooperative concurrency* that occurs when multiple threads communicate to perform a common task. The mainstream solution for cooperative concurrency is coordinated atomic (CA) actions that propose to complement transactions with conversations to provide coordinated error recovery. This approach applies to distributed objects like clients and databases in a message passing context [10], e.g., the systems surveyed in [11] whose distributed modules are presented in [12]. In contrast, our approach targets modern multi-core architectures thus benefiting from shared memory to coordinate efficiently the recovery among concurrent threads. For example, our approach shares the concept of guarded isolated regions for multi-party interactions from [13] without requiring a manager to synchronize the distributed interaction participants. Furthermore, a programmer needs to include a significant amount of code to construct the CA action structure in her program using frameworks specifically designed for this purpose [11, 14], whereas in our approach the programmer can simply relate dependent code regions of an atomic box using built-in language constructs and their parameters.

A more recent checkpointing abstraction for concurrent ML, called *stabilizers*, monitors message receipts and memory updates to help recovering from errors [15]. Stabilizers can cope with transient errors but do not allow coordinated exception handlers to encompass permanent errors.

Failboxes [3] ensure cooperative detection of exceptions in Java. A thread that raises an exception while executing the code encapsulated in a failbox sends a signal to the concurrent threads that are also executing in the same failbox. Upon reception of this signal an exception is raised so that all threads can terminate, which ensures that no thread keeps running on a possible inconsistent shared state. Failbox does not provide coordinated exception handling because the inconsistent state produced by the error cannot be reverted, hence the system has no other solutions but stopping. One could use failboxes to stop the entire concurrent program and restart it manually, however, restarting the program from the beginning may not prevent the same exception to occur again. In contrast, `abox`es automatically rollback their changes upon exceptions and let the programmer define recovery handlers to remedy the cause of an exception and redirect the control flow.

**Transactional memory.** Transactional memory (TM) [4] is a concurrent programming paradigm that lets the programmer delimit regions of code corresponding to *transactions*. A TM ensures that each transaction appears to be executing atomically: either it is aborted and none of its changes are visible from the rest of the system, in which case the transaction can be restarted, or it commits and writes all its changes into the shared memory. The TM infrastructure checks whether memory locations have been accessed by concurrent transactions in such a way that conflicts prevent them from being serialized, i.e., from being executed as if they were sequentially ordered one after the other. In such case, one of the conflicting transactions has to abort.

The inherent isolation of transactions may seem a limitation to achieve high levels of concurrency with some cooperative programming patterns, such as producer-consumer interactions that have inter-thread dependencies. Several contention management policies for TMs have been proposed, however, to alleviate this problem and provide progress guarantees [16, 17]. Indeed a TM conveys a simple rollback mechanism on which one can build coordinated exception handling. While originally proposed in hardware [18], many software implementations of TMs have since been proposed [19–25].

More recently, transactional (atomic) blocks have been suggested as a potential solution for exception handling. Shinnar et al. [26] proposed a `try_all` block for C#, which is basically a `try` block capable of undoing the actions performed inside the block. Cabral and Marques [27] similarly propose to augment the `try` block with transactional semantics (using TM as underlying mechanism) to allow the retry of a `try` block when necessary. Other work proposed richer atomic block constructs that build upon TM and that help with exception handling [28–30]. However, all the existing implementations for the above work focus on sequential executions, hence being unable to cope with coordinated exception handling. When a thread raises an exception, it can either rollback or propagate the exception. If the exception is not caught correctly, the thread may stop and leave the memory in a corrupted state that other threads may access.

## 3 A Running Example

In this section, we introduce an example code (Figure 2) which we later use to explain different aspects of atomic boxes. The example represents a multi-threaded application with a shared task queue `taskQueue` from which threads get tasks to process. All threads execute the same code. Once a thread obtains a task, it first performs pre-computation work (getting necessary inputs and configuring the task accordingly) in the `prepare` method. The execution of the task is performed in the `execute` method of the thread, by calling sequentially the `process` and `generateOutput` methods of the task. We assume that `generateOutput` can add new tasks in the `taskQueue`.

```
public void run() {
  Task task = null;
  while(true) {
    synchronized(taskQueue) {
      task = taskQueue.remove();
    }
    if (task == null) break;
    prepare(task);
    execute(task);
  }
}

public void prepare(Task task) {
  task.getInput();
  task.configure();
}

// No exception handling
public void execute(Task task) {
  task.process();
  task.generateOutput();
}
```

**Fig. 2.** A simple example where multiple threads process tasks from a common task queue and that would benefit from concurrent exception handling.

In what follows, we will mainly focus on the `execute` method of the thread. The code of the method is given without any exception handling. The traditional approach would be to use a `try-catch` statement enclosing the content of the `execute` method. However, when an exception is caught, one cannot easily determine at what point the execution of the method was interrupted and hence, in general, it is difficult to revert to the state at the beginning of the method. In such a case the `task` object could stay in an inconsistent state, possibly even affecting the state shared with other threads, and it would not be possible to simply put the task back into the `taskQueue` for later re-processing. The loss of a task might require other threads to reconfigure, or to stop execution altogether for safety or performance reasons: shared state may be inconsistent, incomplete processing would be worthless. We will see in the next section using this example how atomic boxes prevent the loss of the task and how they allow us to correct the cause of the exception and coordinate threads for the program to recover.

## 4 Syntax and Semantics

Our language extension deals mainly with code blocks that are dependent on each other in the sense that if a statement in one of the blocks raises an exception not handled within the block, none of the other code blocks should continue executing. We call such blocks *dependent blocks.* An atomic box is a group of dependent code blocks that are dependent and can act together to recover from an exception that is raised in at least one of the code blocks. In order to express an atomic box, each dependent code block belonging to an atomic block is enclosed inside a new Java statement, `abox-recover`. The fact that `abox-recover` statements belong to the same atomic box is specified by assigning them the same name. The name of an atomic box is assigned to an `abox-recover` statement as a parameter.

An atomic box can be descendent of another atomic box, which means that the atomic box is dependent on the parent atomic box. Relating an atomic box as a descendant of another atomic box is achieved by assigning a descendant name in the hierarchical naming space. If associated `abox-recover` statements of the same atomic box execute on different threads, these threads are said to be executing in the same atomic box.

Basically, an `abox-recover` statement is composed of two consecutive blocks: the first block is called `abox` and the second `recover`. The precise syntax of the `abox-recover` statement can be described as follows:

**abox** [ (*"name"*, <handlingContext>) ]
**{** S **}**
[ **recover**(ABoxException <exceptionName>)
**{** S' **}** ]

where `abox` and `recover` are keywords, *S* and *S'* are sequences of statements (that may include the additional keywords `retry` and `leave` introduced by our language extension), `name` and `<handlingContext>` are parameters of the associated `abox` keyword and the `<exceptionName>` is the parameter of the `recover` keyword. Optional parameters and structures are enclosed in square brackets: `abox` may have no parameter and the block `recover` is optional.

An `abox` encloses a dependent code block of the application, while the `recover` block specifies how exceptions not caught in the `abox` are handled. If an unhandled exception is raised in an `abox`, we say that the `abox` fails. An `abox-recover` statement provides the convenience of `try-catch` to a dependent block with the following notable differences:

- *Failure atomicity:* An `abox` of an `abox-recover` statement can be rolled back, i.e., either the contents of the `abox` performs all of its modifications successfully (thus none of the `abox`es that belong to the same atomic box fail at any point), or the `abox` acts as if it has not performed any modifications. The failure atomicity property of the `abox` is possible because an `abox` is executed inside a *transaction.*
- *Dependency-safety:* An atomic box ensures dependency safety; i.e., if a statement fails raising an exception, all statements that depend on the failing statement do not execute. The dependency relation between statements is established by naming `abox-recover` statements with a common name (or with names of descendents). The dependency-safety is ensured by two properties of `abox-recover` statement: *i)* An `abox` executes in a transaction,

thus its execution is isolated from all dependent code in the system until it commits. In other words, none of the dependent code blocks see the effects of each other as long as code blocks do not commit. *ii)* If an exception is not handled in an `abox` it rolls back its changes and recovery actions are taken only after all the `abox`es of an atomic box are rolled back. Thus, in no situation it is possible for a dependent code block to see partial modifications of the another dependent block that is in inconsistent state.

– *Coordinated exception handling:* A `try-catch` statement offers a recovery from exception only for the thread on which the exception occurs. The `abox-recover` statement allows the programmer to inform concurrently executing threads of an exception raised in one of the threads. Moreover, through the `recover` block of the `abox-recover` statement it is possible to recover from that exception in a coordinated manner. Note that the coordination is possible among `recover` blocks because they do not execute in a transaction.
– Last, an `abox` and its associated `recover` block can include `try-catch` statements to handle exceptions raised in their context.

We distinguish two different modes of operation for in a `abox-recover` statement: *normal mode* and *failure mode*. The normal mode is associated with `abox` and the failure mode is associated with the `recover` block. An `abox` executes in normal mode, i.e., an `abox` executes as long as no exceptions are raised or until an exception raised inside `abox` propagates outside of the block. Note that if the code inside `abox` raises an exception, and this exception is caught in the block itself, the `abox` still executes in normal mode.

When an exception is propagated out of `abox` boundaries (i.e., when an unhandled exception is raised in the `abox`), the `abox` is said to fail and its `abox-recover` statement switches to failure mode. The failure model of the `abox-recover` statement is such that when the block `abox` fails, its associated atomic box also fails (because the atomic box acts as a single entity upon an exception). Thus, all the `abox-recover` statements associated to the atomic box switch to failure mode upon the failure of an `abox`. The failure of an `abox` also triggers the failure of the descendent atomic boxes.

In the failure mode all the threads that execute in the atomic box coordinate together. They wait for each other to ensure that all the associated `abox-recover` statements switch to failure mode and all the `abox`es are rolled back. Then they perform recovery actions as specified by the `abox` where the exception is raised. After the recovery actions are terminated all the threads decide locally how to redirect their local control flow. There are three options in redirecting the control flow at the end of recovery: restarting, continuing with the statement that comes after `abox-recover` statement, or raising an exception (i.e., abrupt termination). The first two options are provided through two new control flow keywords (`retry` and `leave` respectively), while raising an exception is done by the usual `throw` statement.

In the rest of this section, we detail the constructs for normal and failure modes, the control flow keywords, and the nesting of atomic boxes. We will also discuss the semantics of the `abox-recover` statement under concurrently raised exceptions.

### 4.1 Normal Mode Constructs

The only normal mode construct introduced by our language extension is the `abox`. An `abox` encloses a dependent code block of an atomic box. The block is part of the application code and the fact that it is enclosed in an `abox` does not modify its functionality except for exception handling. In other words, as long as no exception is propagated out of the dependent code block, there is no difference in terms of correctness of the application to have the block in an `abox` or not. However, inserting the code in an `abox` increases safety and provides a means for handling exceptions across multiple threads.

Although the functionality of the code inserted in an `abox` is not modified, an `abox` has different semantics compared to traditional blocks: `abox` executes as a transaction. That way, the modifications performed by the code inside the `abox` are only guaranteed to be effective if the `abox` successfully terminates (hence, if it successfully commits without switching to failure mode). Otherwise none of the modifications performed in the context of the `abox` are visible by code outside the `abox`. Therefore, the code in an `abox` executes atomically and in isolation.

The transactional nature of the `abox` normal execution does not have effect on the correctness of enclosed code but has implications on its execution time. As the transactional execution is provided by a underlying transactional memory (TM) runtime, it incurs two types of latency overhead: *i)* data accesses in the `abox` are under the control of TM and will be slower than bare data accesses; *ii)* in multi-threaded code if different `abox`es concurrently perform accesses on shared data in a way that inconsistencies would occur, an `abox` may be aborted, rolled back and restarted, which adds extra latency to its execution.

In its simplest form (i.e., when its optional parameters are omitted) the syntax for an `abox` is

**abox {** S **}**

where $S$ is a sequences of statements. The statements in $S$ may contain traditional Java statements as well as the control flow keywords added by our language extension (see Section 4.3). For the sake of simplicity, in this paper we do not consider Java statements that perform irrevocable actions (e.g., I/O operation or system calls) in an `abox` because most underlying TM implementations do not support transactional execution for such actions. There exist however practical solutions to this limitation (e.g., in [31, 32]).

The simplest form of an `abox` is considered as an indication that the block is the only block in an atomic box, and thus it does not have any dependencies on other parts of the code. For such `abox` the exception handling is done locally without any coordination with any other `abox`. Hence, this form is suitable for exception handling in single-threaded applications as well as handling of exceptions for code blocks of multi-threaded applications that do not have any implications on other running threads.

As an example of such scenario, assume that an `OutOfMemoryError` is raised during the execution of the `execute` method of Figure 2. If for the running multi-threaded application, it is known that most of the tasks has small memory footprint but occasionally some tasks can have large memory footprint (but never exceeding the heap size allocated by the JVM), it is possible to clean up some resources or wait for a while before restarting execution. This would solve the problem if memory is freed when a task with a possibly large footprint finishes executing. Using the simple form of `abox`, the code for this solution would be as in Figure 3 (the syntax for the `recover` block will be explained shortly). Note

```
public void execute(Task task) {
  abox {
    task.process();
    task.generateOutput();
  } recover(ABoxException e) {
    if(e.getCauseClass() == OutOfMemoryError.getClass()) {
      // Back off (sleep) upon OutOfMemoryError
      backOff();
    }
    // Implicit restart
  }
}
```

**Fig. 3.** Local recovery for an `OutOfMemoryError` using the simple form of `abox`.

that this solution is not possible with either a `try-catch` block or a failbox since the state of the `task` object cannot be rolled back to its initial state.

A programmer can describe an atomic box composed of multiple `abox`es by assigning all of the associated `abox`es the same name. The syntax for expressing an `abox` of such an atomic box is:

$$\textbf{abox}(\textit{"name"}, \langle\text{handlingContext}\rangle)\ \{\ \text{S}\ \}$$

where the `name` parameter is a string that associates the `abox` to the atomic box it belongs to, and the `<handlingContext>` parameter is a keyword describing which `recover` blocks will execute for performing recovery. Since the `<handlingContext>` parameter effects the execution of `recover` block, details on this parameter are provided with the description of `recover` block in Section 4.2.

Contrarily to the simplest form of `abox`, the named form implies that upon failure of the `abox` the exception handling should be coordinated across the atomic box. This form serves mostly in handling exceptions in multi-threaded applications.

We can slightly change the conditions to the example for which `abox` provided a solution in Figure 3 and generate a different scenario. Let us assume that in the example there are not many solutions for solving the `OutOfMemoryError` and the programmer simply wants to stop all the threads when such an exception is raised. The code that will provide this solution would be as in Figure 4.

```
public void execute(Task task) {
  abox("killAll", all) {
    task.process();
    task.generateOutput();
  } recover(ABoxException e) {
    if(e.getCauseClass() == OutOfMemoryError.getClass()) {
      // Upon OutOfMemoryError, propagate to terminate thread
      throw e;
    }
  }
}
```

**Fig. 4.** Coordinated termination of a multi-threaded application upon an `OutOfMemoryError`. The named form of `abox` can be used to provide such recovery.

Note that all the threads are running the same code. The code in Figure 4 uses the named form of `abox`. The `<handlingContext>` parameter is given as `all`, which means that when the `OutOfMemoryError` is raised on one thread, all the threads running in the atomic box will execute their `recover` blocks. In the `recover` block an exception is raised so that the currently executing thread dies (since the threads are assumed to be running the code in Figure 2, the exception will not be caught and each thread will be terminated). This solution is again not possible with a `try-catch` statement. Since the objective in this example is to stop the application, the failbox approach would also work: one could enclose the content of the `execute` method in an `enter` block, which would specify that the code enters a failbox common to all threads.

We can also think about a variant of the above example that cannot be resolved using the failbox approach. Let us assume that, as the `task` object can configure itself before execution, it is also possible to reconfigure it to perform the same job using less memory but slower (e.g., by disabling an object pool). In such a case, the named form of the `abox` allows us to resolve the problem with the code in Figure 5 (again only by changing the content of the execute method). This solution is possible with the named form of `abox` since the `abox-recover` statement including the `abox` provides failure atomicity and coordinated exception handling. The failure atomicity of the property of the `abox-recover` statement allows the modifications of the execution inside the `abox` to be rolled back, thus the task object can be reverted to a consistent state where it can be reconfigured. The coordinated exception handling provided by the `abox-recover` statement

```
public void execute(Task task) {
  abox("reconfigure", all) {
    task.process();
    task.generateOutput();
  } recover(ABoxException e) {
    if(e.getCauseClass() == OutOfMemoryError.getClass()) {
      // Upon OutOfMemoryError, reconfigure and restart
      task.reconfigure();
    }
  }
}
```

**Fig. 5.** Coordinated recovery to reconfigure tasks (for decreasing their memory footprint) upon `OutOfMemoryError`.

allows the same behavior to be performed on all threads in a synchronized way and remedy the problem in a single step.

### 4.2 Failure Mode Constructs

Since an atomic box corresponds to dependent code blocks, when an `abox` fails, its associated atomic box also fails. We call the atomic box that fails upon the failure of an `abox` an *active* atomic box. An active atomic box is defined as the set of `abox`es of the same atomic box that have started executing and that have not yet started committing. This set is defined as long as at least one thread executes in the atomic box.

We argue that in terms of failure it is enough to consider an active atomic box rather than all the statically defined atomic box to ensure dependency-safety and failure atomicity. Since `abox`es that have started committing are guaranteed not to execute on any inconsistent state that can be generated by the `abox`es of the active atomic box (`abox`es execute in isolation), their exclusion does not harm dependency-safety. Moreover, the consistency of data is ensured as long as the commit of `abox`es that have started committing are allowed to finish before the `abox`es of the active atomic box start performing recovery actions. So the rollback of an active atomic box does not require `abox`es that have already started committing to rollback. Hence, it is safe to provide failure atomicity only for an active atomic box.

To have better understanding of the concept of active atomic box consider the solution proposed in Figure 4. For this solution if we think that the tasks executed by all of the threads have more or less the same load, the threads will generally be executing the `execute` method at about the same time periods. However, if we think of a scenario where tasks have variable load, this may not be true. So when the `OutOfMemoryError` is raised, some threads may be executing in the content of the `abox`, while some others may be still committing the `abox` in the `execute` method and some others maybe fetching a new task from the taskQueue (these threads have not yet entered in an abox). In such a case, the proposed solution may not stop all the threads since not all may be executing in the active atomic box when the OutOfMemoryError is raised. However, for these non-terminated threads the execution continues safely; threads that were committing while the exception is raised in active atomic box do not have any more dependence on the aboxes of the atomic box, and threads that have not yet entered execution in the atomic box may not raise an OutOfMemoryError if there is enough memory once the threads of the active atomic box get killed. Even if an OutOfMemoryError is again raised, this will be resolved by the active atomic box defined at the time of the second exception. Hence, we see that by applying the failure atomicity and dependency-safety only on the active atomic box it is also possible to provide safe executions.

The failure of an active atomic box results in the following coordinated behavior in the aboxes that constitute the active atomic box:

1. The aboxes that constitute an active atomic box switch to failure mode. This triggers the coordinated failure behavior of the atomic box.
2. All the aboxes that switch to failure mode automatically rollback. At the same time all aboxes that have started committing terminate their commit.
3. All the threads executing in an active atomic box are notified of a special exception ABoxException (the structure of this exception is explained later).
4. All the threads executing in an active atomic box wait for each other to make sure that they all rolled back and received the ABoxException notification. The threads in the active atomic box also wait for threads running an abox that have already started committing to finish their commit operation (which may not succeed and trigger an abort).
5. All the aboxes that constitute an active atomic box perform the recovery actions in the associated recover blocks according to the ABoxException they receive. Entry in the atomic box is forbidden for any thread during recovery.
6. All the threads executing in an active atomic box wait for each other to terminate their recovery actions. Once all recovery actions are terminated each of the threads executing in the active atomic box decide locally how to redirect their control after failure.

***The ABoxException.*** The structure of the ABoxException that is notified to all the threads in the active atomic box is as follows:

```
public class ABoxException {
  Class causeClass;
  String message;
  Thread source;
  String aboxName;
  int handlingContext;
  // Methods omitted...
}
```

where the `causeClass` field stores the class of the exception raised by the `abox` that initially failed (initiator `abox`), the `message` field is the message of the original exception, the `source` field is the reference to the `Thread` object executing the initiator `abox`, `aboxName` is the name of the failing atomic box and `handlingContext` is an integer value that defines which of the corresponding `recover` blocks associated to the atomic box will be executed. The value of the `handlingContext` corresponds to the `<handlingContext>` parameter of the initiator `abox` (the details for the values of `handlingContext` are explained below together with the `recover` block). Note that the `ABoxException` stores the class of the original exception object that initiated the atomic box failure rather than its reference. This is a deliberate choice since the original exception object can include references to other objects that are allocated inside the initiator `abox` and that will be invalidated by the rollback performed upon the failure of the atomic box.

***The recover block.*** A `recover` block encloses recovery actions to be executed when the `abox` it is associated to fails. Since the `recover` block is related to failure of an atomic box, it is only part of failure mode execution. Note also that the `recover` block does not execute in a transactional context; it always executes after its corresponding `abox` rolls back. The decision of whether the `recover` block will be executed depends on the `handlingContext` parameter of `ABoxException` sent by the initiator `abox`. Two values exist for the parameter `handlingContext`: `local` and `all`. With the `local` option, only the `recover` block of the initiator `abox` will be executed, other threads will not execute any recovery action. If the `all` option is chosen all the threads executing in the atomic box execute their respective `recover` blocks.

Whichever of the `handlingContext` options is chosen, once the `recover` block executions are terminated each of the threads executing in the atomic box take their own control flow decision. If the `handlingContext` parameter has the value `local`, the initiator `abox` redirects the control flow according the control flow keyword used in its `recover` block (for the control flow keywords see Section 4.3). All the other threads in the atomic box re-execute the `abox` for which they perform recovery actions. If the `handlingContext` parameter has the value `all`, each of the threads redirects the control flow according the control flow keyword used in its respective `recover` block.

If the `recover` block of `abox-recover` statement has been omitted, the thread executing this `abox-recover` statement performs no recovery and re-executes the `abox` of the `abox-recover` statement.

The syntax of the `recover` block can be described as follows:

$$\textbf{recover}(\text{ABoxException exceptionName})\ \boldsymbol{\{}\ \text{S}\ \boldsymbol{\}}$$

where the `exceptionName` is the name of the `ABoxException` notified to all the threads upon failure of an atomic box. The exception parameter of the `recover` block is expected to be of type `ABoxException` and providing an exception of another type will produce a compiler error.

Having analyzed most of the properties of the normal and failure modes, it would be appropriate to analyze the mechanisms described above in an example. At this point we can use another variant of the running example of Figure 2 with an `OutOfMemoryError` being raised during the execution of the `execute` method. Suppose, in this case, that the programmer knows that he is using too many threads and if the heap allocated by the JVM is not enough, it would be enough for him to kill only some of the worker threads. This would effectively handle the exception while keeping the parallelism of thread execution at a reasonable level. Since the programmer would not know the size of the memory allocated in advance he can choose to implement the solution in Figure 6 using the atomic boxes.

```
public void execute(Task task) {
  abox("killSome", local) {
    task.process();
    task.generateOutput();
  } recover(ABoxException e) {
    if(e.getCauseClass() == OutOfMemoryError.getClass()) {
      // Upon OutOfMemoryError, propagate to terminate local thread
      throw e;
    }
  }
}
```

**Fig. 6.** Coordinated recovery to decrease the memory used by the multi-threaded application by only killing some of the threads upon `OutOfMemoryError`.

The solution shown in Figure 6 is the same as the code in Figure 4 except that the name of the <handlingContext> parameter is set to `local` instead of `all`. With this change each time an `OutOfMemoryError` is raised only the thread raising the exception executes the `throw` statement and kills itself. This solution works better than a simple `try-catch` because with the `try-catch` solution multiple threads could have raised the same exception at the same time and, being unaware of the exceptions raised in other threads, all of these threads would kill themselves leaving a smaller amount of threads running in the system, rather than gradually decreasing the amount of concurrency. Gradual decrease is possible thanks to the coordinated nature of the exception handling: coordination imposes the threads to abort their `abox`es (instead of killing themselves) and restart execution after the initator `abox`'s thread is killed. Thanks to the failure atomicity provided by atomic boxes, this can safely be repeated as many times as required until the required number of threads are killed.

### 4.3 Redirecting Control Flow after Recovery

For providing control flow specific to `abox-recover` statement, we introduce two new control flow keywords: `leave` and `retry`. These keywords are to be used mainly inside `recover` blocks but they can also be used with similar semantics also in the `abox`es. The only difference of using the keywords in an `abox` is that they immediately fail the `abox` (and respectively also the active atomic box) and they behave as a `recover` block that has no other recovery actions but only the specified keyword. Thus, the existence of these keywords in the `abox` will just serve as a shortcut to a case where the atomic block has failed and we execute only a `leave` or `retry` inside the `recover` block.

If no control flow keyword is provided, upon exit the `recover` block implicitly re-execute the associated active `abox`. A programmer can also explicitly ask for re-execution of the associated `abox` using the `retry` keyword. In contrast, a `leave` keyword will pass the control to the statement following the `recover` block. Note that with a `leave` keyword, the effect of an `abox` is as if it had never executed. The reason is that the failure of the `abox` has caused the rollback of the modifications performed within.

The use of `throw` statement inside `recover` block will quit the `recover` block and propagate the exception in the context of the statement following the `recover` block. With a `throw` statement, again the atomic box appears as if it has never executed. Similarly if an unhandled exception is raised in recovery action code enclosed in a `recover` block, the behavior is the same as an explicit `throw` statement.

Any already existing control flow keyword (except the `throw` keyword) that quits a block (i.e., `continue`, `break` and `return`) does not change semantics with our language extension. When used inside an `abox` (and not used inside a nested block such as a loop) they imply immediate commit of the tentative modifications up to the point of occurrence of the keyword and pass the control to the target destination outside the `abox` and `recover` block. If those control keywords are used inside a `recover` block, they behave exactly the same way as in the `abox` except that, since the `abox` is rolled back, none of the effects of the `abox` are visible (but of course the modifications inside the `recover` block are effective).

The use of a `throw` statement inside the `abox` raises an exception in the block as in plain Java. If the exception is handled inside the `abox` the behavior of the `throw` statement is unchanged. However, if the exception is not handled in the `abox`, the `abox` (and the corresponding active atomic box) switches to failure mode.

### 4.4 Nesting of Atomic Boxes

The failure of an `abox` can also trigger the failure of an atomic box other than the one it belongs to. If the failing atomic box is parent of another atomic box, when the parent atomic box fails, the child atomic box also fails, thus both the parent and the child atomic boxes switch to failure mode. In contrast, when a child atomic box fails, its parent atomic box does not fail, thus the child atomic box switches to failure mode, while the parent atomic box does not.

The fact that atomic boxes have ascendants or descendants is reflected by a hierarchical naming of `abox`es. The `name` parameter of an `abox` can be a string of the form `x.y.z` following the naming convention of Java package names.

### 4.5 Resolution of Concurrently Raised Exceptions

Up to this point we have considered only the case where a single `abox` initiates an atomic box failure. If an exception needs to be treated by an `abox`, this is most probably because the exception concerns all the threads executing in the atomic box. So it is not surprising to expect that multiple `abox`es raise the same exception and fail the atomic box. It is also perfectly possible that different `abox`es of the same atomic box, concurrently raise the different exceptions and cause the atomic box to fail.

The atomic box takes a very simple approach to resolve concurrently raised exceptions thanks to its failure atomicity property: an atomic box allows only one exception (the first one to be caught) to be treated in failure mode and ignores all the concurrently raised exceptions during failure mode.

The atomic box does not consider all the concurrently raised exceptions together. By handling one exception and removing its cause before re-execution, one may avoid other concurrent exceptions to occur again. During re-execution, if the cause of the concurrently raised exceptions are not removed they will again manifest and fail the atomic box. They will thus be treated during re-execution.

As can be noticed, among other advantages, the atomic box approach brings an elegant solution to the concurrent exception handling problem thanks to its failure atomicity property. Actually, the solution presented in Figure 6 is a good example illustrating the resolution of concurrently raised exceptions. In this example, other than the coordinated nature of the exception handling, it is the simple concurrent exception handling approach taken by atomic boxes that allows us to kill only as many threads as required.

## 5 Atomic Boxes Implementation

We have implemented a concurrent exception handling compiler framework, called CXH, that supports the language constructs proposed in Section 4. The CXH compiler framework produces bytecode that is executable by any Java virtual machine in a three-step process. First it runs our pre-compiler, TMJava that converts the extended language into annotated Java code. The annotations are used to detect in the bytecode, which parts of the code have the `abox` semantics. Second our CXH embeds the LSA transactional memory library [24] that provides wrappers to shared memory accesses. Our `abox`es benefit from the speculative execution of TMs to ensure that no exceptions are raised before applying any change in the shared memory. Third, CXH uses an existing bytecode instrumentation framework, Deuce [33], that redirects calls within annotated methods to transactional wrappers. We describe below these three components in further detail.

### 5.1 Language Support for Atomic Boxes

We implemented TMJava, a Java pre-compiler that converts `abox-recover` constructs in annotated Java code. This allows us to compile the resulting code using any Java compiler. TMJava converts each `abox` into a dedicated method that is annotated with an `@Atomic` keyword. More precisely, TMJava analyzes the code to find the `abox`es (`abox` keyword) inside class methods. Then, for each such `abox` it creates a new method whose body is the content of the corresponding `abox` and replaces the original `abox` with a call to this new method. The conversion of an `abox` $a$ into a method $m$ requires passing some variables to the produced method $m$ to address the following issues:

1. Variables that belong to the context of the method enclosing the `abox` $a$ should also be accessible inside the scope of the produced method $m$.
2. Variables that belong to the context of the method enclosing the `abox` $a$ and that are modified inside $a$ should have their modifications effective outside the produced method $m$ (as it would be for `abox` $a$).

To ensure that variables are still visible inside the produced methods, the variables whose scope are out of `abox` context are passed as input parameters to the corresponding method. For the state of variables to be reflected outside the scope of the `abox`, these variables are passed as parameters using arrays (if the variables are of primitive types). When the method returns, we copy back these array elements into the corresponding variables.

The resulting annotated Java code can be compiled using any Java compiler. TMJava is available for download from `http://tmware.org/tmjava`.

### 5.2 Transactional Memory Wrappers

We use LSA [24], an efficient time-based transactional memory algorithm that maps each shared memory location with a timestamp. Each transaction of LSA executes speculatively by buffering its modifications. If the transactions reaches its end without having aborted, it attempts to commit by applying its modifications to shared memory. More precisely, when a transaction starts it records the value of a global time base, implemented as a shared counter. Upon writing a shared location, the transaction acquires an associated ownership record, buffers the write into a log, and continues executing subsequent accesses. At the end, when the transaction tries to commit, it reports all the logged writes in memory by writing the value, incrementing the global counter, and associating its new version to all written locations as part of the ownership records. Upon reading a shared location, it first checks if the location is locked (and aborts if locked), then compares the version of the location to the counter value it has seen. If the location has a higher version than this value, this means that a concurrent transaction has modified the location, indicating a conflict.

The particularity of the LSA algorithm is to allow the transaction to commit despite such a conflict thanks to incremental validation: if all previously read values are still consistent, i.e., their versions have not changed since they have been read, the transaction has a valid consistent snapshot and can resume without aborting.

Our `abox` leverages memory transactions that execute speculatively on shared data. The main difference between `abox`es and the transactions lies in the fact that each `abox` decides whether to abort or commit its changes also depending on (concurrent) exceptions raised. Before committing, an `abox` makes sure that no exception was raised inside the block or by a dependent `abox`.

### 5.3 Bytecode Instrumentation

After compilation we obtain a bytecode where annotated methods directly access the memory. To ensure that these annotated methods, which correspond to the original `abox`es, execute speculatively we have to redirect their memory accesses to the transactional memory. To that end, we use the Deuce framework [33] to instrument the annotated method calls at load time. Deuce instruments class methods annotated with `@Atomic` such that accesses to shared data inside those methods are performed transactionally. This bytecode instrumentation redirects all `abox` memory accesses to LSA so that each `abox` executes as a transaction.

## 6 Evaluation

We compare our `abox` solution against failbox [3] on an Intel Core2 CPU running at 2.13GHz. It has 8-way associative L1 caches of 32KB and an 8-way associative L2 cache of 2MB. For `abox` we implemented the compiler framework as explained in Section 5 whereas for failboxes we reused the original code from [3].

### 6.1 Producer-Consumer Example

Our first experiments consist of a simple producer-consumer application, where one thread pushes an item to a shared stack while another pops the topmost item from the same stack. For the sake of evaluation, the stack `push()` method raises an exception if adding the new item to the stack would exceed its capacity. We evaluated two versions of the same program: one using failbox, the other using our `abox`. The execution time of these two versions has been evaluated in normal cases (where we fill the stack prior to execution such that no exceptions are raised) and for handling exceptions (where we try to push an item to an already full stack). Results are averaged over 100 executions.

Tables 1 and 2 report the minimum, maximum and average execution time in microseconds, respectively without and with exceptions. On the one hand, we observe that our solution executes about 2× faster (on average) than failboxes in normal executions. This is due to a cache effect observed with failbox approach. Each time a failbox is entered a shared variable is checked to verify whether it

| | min | max | average |
|---|---|---|---|
| `abox` | 7.27 | 11.67 | 8.92 |
| failbox | 15.70 | 34.97 | 18.58 |
| speedup of `abox` | 1.34 | 4.81 | 2.08 |

**Table 1.** Execution times of `abox` and failbox (in microseconds) on a multi-threaded producer-consumer application when no exception is raised.

has failed. Since this experiment requires very frequent entries to a failbox by multiple threads the failbox entries are serialized. Our implementation does not suffer from this problem since the check for the failure of an `abox` does not need to be verified often (an `abox` is executed in isolation from other code).

On the other hand, our solution performs more than 15× faster (on average) than failboxes to handle exceptions. We conjecture that it is due to the fact that failbox approach uses the `interrupt` mechanism to communicate the exception on one thread to the other threads. The `abox` approach communicates over the shared memory, resulting in a faster notification. It is worth mentioning that our `abox`es permit both `push()` and `pop()` methods to recover from exception, allowing the program to resume, while failbox simply stops the program upon the first exception raised. Considering this desirable behavior and the observed overhead, `abox` clearly represents a promising approach.

| | min | max | average |
|---|---|---|---|
| `abox` | 1.40 | 2.62 | 2.22 |
| failbox | 32.167 | 47.23 | 34.55 |
| speedup of `abox` | 12.28 | 33.74 | 15.7 |

**Table 2.** Execution times of `abox` and failbox (in microseconds) on a multi-threaded producer-consumer application when exceptions are raised.

### 6.2 Sorting Examples

Our second experiments rely on two single-threaded sorting applications (quick-sort and bubble-sort) coded in 3 ways: *(i)* using plain Java (with no extensions), *(ii)* inside failboxes, and *(iii)* inside `abox` blocks. The plain Java version is used to measure the inherent overhead of failbox and `abox` versions. The sort is performed inside a function and the application can choose to run either a *quick-sort* or a *bubble-sort* function.

Figures 7 through 9 depict the performance of failbox and `abox` on quick-sort (left column) and bubble-sort (right column). Figure 7 compares the execution overhead due to entering and leaving an `abox` block or a failbox (we call this *begin/end overhead*). Figure 8 shows the execution time performance of `abox` and

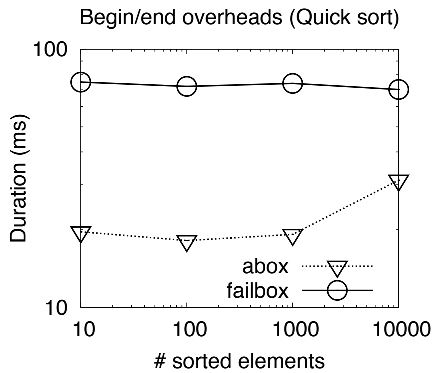


**Fig. 7.** Comparison of the overhead produced when starting and terminating an `abox` and a failbox (note the logarithmic scales on both axes).

failbox executions without the *begin/end overhead*. Figure 9 depicts the total execution time performance of `abox` and failbox. The execution time performance depicted in figures 8 and 9 are given as the slowdown with respect to the performance of the plain Java version, which does not have any begin/end overhead. Each point in the graphs corresponds to the average of 10 runs.

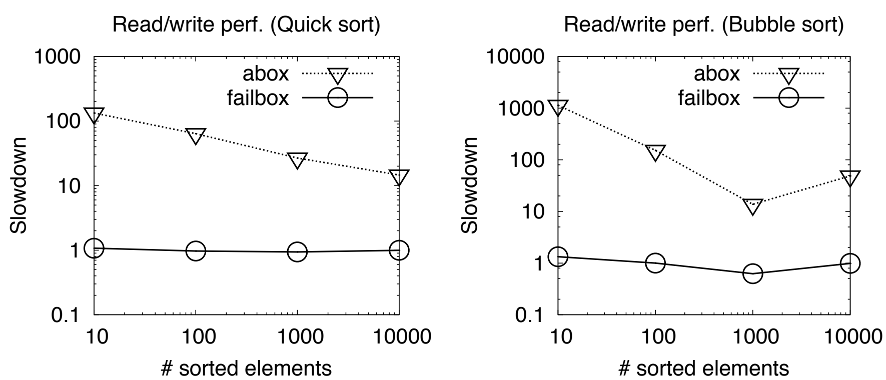


**Fig. 8.** Comparison of the overhead due to accessing the shared memory in `abox` and failbox (note the logarithmic scales on both axes).

The results show that although the failbox approach performs as good as plain Java inside the failbox, its begin/end overhead is quite high. We attribute this high overhead of the failbox approach to the memory allocation performed

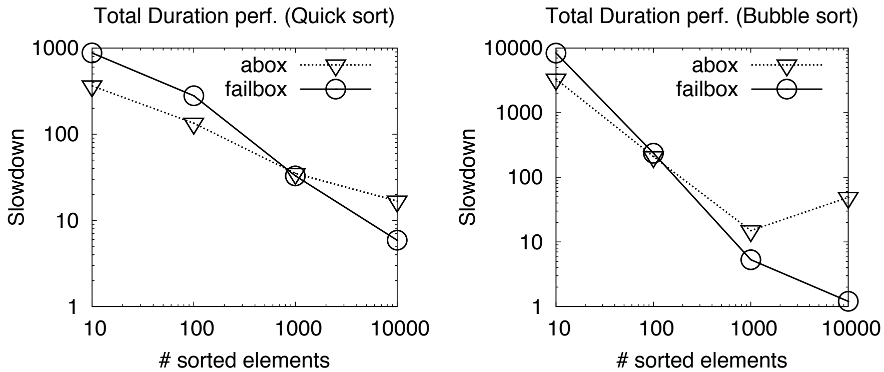


**Fig. 9.** Comparison of the total duration time of `abox` and failbox (note the logarithmic scales on both axes).

to generate a new failbox (be it a child or a new failbox) before entering the failbox. Figure 9 also illustrates that `abox` blocks perform better than the failbox approach for input arrays of up to about 1000 elements. This demonstrates that our `abox` implementation, although using transactions to sort array elements, performs well even compared to simpler approaches that do not roll back state changes.

## 7 Conclusion and Future Work

This paper introduces language constructs for concurrent exception handling, a way to handle exceptions in a concurrent manner for multi-threaded software.

The key novelty is to ensure that any inconsistent state resulting from an exception cannot be accessed by concurrent threads, thus allowing the programmer to define concurrent exception handlers. The alternative failbox [3] language construct that prevents threads from running on inconsistent states simply stops all threads. Letting the programmer define concurrent exception handlers allows us to recover rather than stop. For example, the programmer can remedy the cause of an exception and retry the concurrent execution.

To experiment with our solution, we have implemented a compiler framework, CXH, for our language constructs that converts `abox`es into code that uses an underlying software transactional memory runtime. Our preliminary evaluations indicate that the overhead of our transactional wrappers is low: when accessing up to hundreds of elements, `abox`es execute twice faster than failboxes.

The fact that the transactional memory overhead does not significantly impact the concurrent exception handling should encourage further research in this direction. This work could for example benefit from ongoing progress in hardware and hybrid transactional memory to further reduce overheads, as our current implementation is purely software based. Even though there is a long road before integrating such language constructs in Java, we believe that exploring transactional memory as a building block for concurrent exception handling will raise new interesting research challenges and offer new possibilities for programmers.

**Acknowledgements**

This work is supported in part by the Swiss National Foundation under grant 200021-118043 and the European Union's Seventh Framework Programme (FP7/2007-2013) under grants 216852 and 248465.

## References

1. Cabral, B., Marques, P.: Exception handling: A field study in java and .NET. In: ECOOP. Volume 4609 of LNCS. (2007) 151–175
2. Stelting, S.: Robust Java: Exception Handling, Testing and Debugging. Prentice Hall, New Jersey (2005)
3. Jacobs, B., Piessens, F.: Failboxes: Provably safe exception handling. In: ECOOP. Volume 5653 of LNCS. (2009) 470–494
4. Harris, T., Larus, J.R., Rajwar, R.: Transactional Memory. 2nd edn. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers (2010)
5. Spector, A.Z., Daniels, D.S., Duchamp, D., Eppinger, J.L., Pausch, R.F.: Distributed transactions for reliable systems. In: SOSP. (1985) 127–146
6. Liskov, B.: Distributed programming in Argus. Commun. ACM **31** (March 1988) 300–312
7. Haines, N., Kindred, D., Morrisett, J.G., Nettles, S.M., Wing, J.M.: Composing first-class transactions. ACM Trans. Program. Lang. Syst. **16** (1994) 1719–1736
8. Jimenez-Peris, R., Patino-Martinez, M., Arevalo, S., Peris, R.J., Ballesteros, F., Carlos, J.: Translib: An ada 95 object oriented framework for building transactional applications (2000)
9. Kienzle, J., Romanovsky, A.: Combining tasking and transactions, part II: open multithreaded transactions. In: IRTAW '00. (2001) 67–74
10. Xu, J., Randell, B., Romanovsky, A., Rubira, C.M.F., Stroud, R.J., Wu, Z.: Fault tolerance in concurrent object-oriented software through coordinated error recovery. In: FTCS. (1995) 499–508
11. Capozucca, A., Guelfi, N., Pelliccione, P., Romanovsky, A., Zorzo, A.F.: Frameworks for designing and implementing dependable systems using coordinated atomic actions: A comparative study. J. Syst. Softw. **82** (2009) 207–228
12. Beder, D., Randell, B., Romanovsky, A., Rubira, C.: On applying coordinated atomic actions and dependable software architectures for developing complex systems. In: ISORC. (2001) 103 –112
13. Zorzo, A.F., Stroud, R.J.: A distributed object-oriented framework for dependable multiparty interactions. In: OOPSLA. (1999) 435–446
14. Filho, F., Rubira, C.F.: Implementing coordinated error recovery for distributed object-oriented systems with AspectJ. J. of Universal Computer Science **10**(7) (2004) 843–858
15. Ziarek, L., Schatz, P., Jagannathan, S.: Stabilizers: A modular checkpointing abstraction for concurrent functional programs. In: ICFP. (2006) 136–147
16. Scherer III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: PODC. (2005)

17. Guerraoui, R., Herlihy, M., Kapalka, M., Pochon, B.: Robust contention management in software transactional memory. In: SCOOL. (2005)
18. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: ISCA. (1993) 289–300
19. Shavit, N., Touitou, D.: Software transactional memory. In: PODC. (1995) 204–213
20. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC. (2003) 92–101
21. Harris, T., Fraser, K.: Language support for lightweight transactions. In: OOPSLA. (2003) 388–402
22. Dalessandro, L., Marathe, V., Spear, M., Scott, M.: Capabilities and limitations of library-based software transactional memory in C++. In: Transact. (2007)
23. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: DISC. Volume 4167 of LNCS. (2006) 194–208
24. Riegel, T., Felber, P., Fetzer, C.: A lazy snapshot algorithm with eager validation. In: DISC. Volume 4167 of LNCS. (2006) 284–298
25. Felber, P., Gramoli, V., Guerraoui, R.: Elastic transactions. In: DISC. Volume 5805 of LNCS. (2009) 93–107
26. Shinnar, A., Tarditi, D., Plesko, M., Steensgaard, B.: Integrating support for undo with exception handling. Technical Report MSR-TR-2004-140, Microsoft Research (2004)
27. Cabral, B., Marques, P.: Implementing retry - featuring AOP. In: Fourth Latin-American Symposium on Dependable Computing. (2009) 73–80
28. Harris, T.: Exceptions and side-effects in atomic blocks. Sci. Comput. Program. **58**(3) (2005) 325–343
29. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: PPoPP. (2005) 48–60
30. Fetzer, C., Felber, P.: Improving program correctness with atomic exception handling. J. of Universal Computer Science **13**(8) (2007) 1047–1072
31. Volos, H., Tack, A.J., Goyal, N., Swift, M.M., Welc, A.: xCalls: safe I/O in memory transactions. In: EuroSys. (2009) 247–260
32. Porter, D.E., Hofmann, O.S., Rossbach, C.J., Benn, A., Witchel, E.: Operating system transactions. In: SOSP. (2009) 161–176
33. Korland, G., Shavit, N., Felber, P.: Deuce: Noninvasive software transactional memory in Java. Transactions on HiPEAC **5**(2) (2010)