Exceptions and Transactions in C++ *

Ali-Reza Adl-Tabatabai[2] Victor Luchangco[3] Virendra J. Marathe[3]
Mark Moir[3] Ravi Narayanaswamy[2] Yang Ni[2] Dan Nussbaum[3]
Xinmin Tian[2] Adam Welc[2] Peng Wu[1]

[1]IBM Research [2]Intel Corporation [3]Sun Microsystems Laboratories

{ali-reza.adl-tabatabai,ravi.narayanaswamy,yang.ni,xinmin.tian,adam.welc}@intel.com
{victor.luchangco,virendra.marathe,mark.moir,dan.nussbaum}@sun.com
pengwu@us.ibm.com

Abstract

There has been significant discussion—and significant disagreement—on the issue of how exceptions should interact with atomic blocks implemented using transactions. We present a proposal that offers a significant contribution towards resolving this issue, at least for C++, and we raise remaining areas of disagreement for discussion at the workshop and in the community in general.

1. Introduction

Transactional memory (TM) [4] is a promising technology for making concurrent programming of multicore computers tractable for everyday programmers. With TM, a thread can execute a *transaction*, consisting of multiple memory accesses that are performed *atomically* (i.e., the accesses appear to happen all at once, or not at all). A programmer specifies *what* should happen atomically, by enclosing it in an *atomic block*, and the system (designer) is responsible for *how* this is achieved, relieving programmers of a significant burden.

To transition from research to practice, TM must be integrated into real, widely used languages. Recognizing this need, TM researchers at IBM, Intel, and Sun have been discussing how to integrate TM into C++. To date these discussions have aimed to agree to the extent we can on a specification, and to understand positions on issues that are difficult to resolve.

We have largely agreed on basic features. Two features used in this paper are *atomic blocks*, which demarcate blocks of code to be executed atomically, and the ability to explicitly *abort* an atomic block. The following example executes the code in `<stmts>` atomically, but if `cond` evaluates to true, the atomic block is aborted and thus has no effect:

```
__tm_atomic {
  <stmts>
  if (cond) {
    __tm_abort;
  }
}
```

How exceptions should interact with atomic blocks has been the subject of active debate among TM researchers, with problems raised for every proposed solution [1, 2, 7]. Although we have not yet reached agreement on this issue, we have developed a proposal for integrating exceptions and atomic blocks in C++ that represents significant progress towards agreement. In this paper, we describe this proposal and highlight for discussion some of the remaining issues.

Exception handling is an error-prone aspect of C++ programs, requiring carefully defined and documented programming conventions. Exceptions prevent a code block from executing completely and thus a thrown exception may leave the program in an indeterminate state. Good C++ programming practices dictate that if an exception crosses the interface of a component, the programmer should strive to provide an *exception safety guarantee* on the state of that component: A *no-throw* guarantee means that the code will never throw an exception. A *strong* guarantee means that the code will appear to have no effect if it throws an exception, typically implemented via catch clauses that explicitly undo side effects. And a *basic* guarantee means that the code still maintains invariants and leaks no resources if it throws an exception that prevents it from executing completely.

C++ conventions encourage programmers to document exception safety guarantees and to provide at least the basic guarantee. For example, in the standard template library (STL) each interface function provides and requires certain levels of exception safety (e.g., destructors and `swap()` operations should have no-throw guarantees).

The same exception safety issues arise for atomic blocks when an exception propagates out of an atomic block. Moreover, concurrency exacerbates exception safety issues as the programmer must now preserve invariants in the presence of concurrent accesses. Consider the following simple example:

```
__tm_atomic {
  X++;
  foo();
  Y++;
}
```

In this example, the programmer wants to maintain the invariant `X==Y` as pre- and post-condition of the atomic block. But what if `foo()` throws an exception? The exception *escapes* the atomic block, so we must determine the fate of the atomic block. One simple answer is that it commits: exceptions are merely an alternate mechanism that causes control to exit a block. This *commit-on-escape* semantics is easy to specify, and the behavior it allows is a restriction of the behavior allowed for the same program without atomic blocks.

Committing this atomic block on an exception, however, may expose broken invariants to other threads. To avoid this, the programmer must catch exceptions and restore invariants within the atomic block, before an exception propagates out of the block.

Commit-on-escape semantics undermines one of the key advantages of TM: It impedes the programmer's ability to reason about an atomic block as if it executes atomically, in an *all-or-nothing* fashion. This diminishes TM's value proposition of simplifying concurrent programming. Commit-on-escape also does not exploit TM's inherent ability to rollback a transaction, a capability that's clearly useful for implementing strong exception safety guarantees.

Because of these issues, some have advocated aborting an atomic block when an exception escapes it, exploiting the transactional mechanism to restore all invariants by rolling back any changes made by the atomic block. With this *abort-on-escape* semantics, the programmer need not worry about partial execution of atomic blocks: the system automatically provides strong exception safety guarantees for atomic blocks. Atomic blocks now have all-or-nothing semantics.

Abort-on-escape poses some tricky semantic questions, however, because the effects of the code that caused the exception are rolled back. In particular, if the exception being thrown was allocated or initialized within the atomic block, then naively aborting it would roll back the allocation and the values written into the exception, leaving the question of what exception would be caught outside the block. While some possibilities exist to attempt to address this issue, even those of us who had favored abort-on-escape have found them unsatisfying.

We could exclude the exception from rollbacks caused by aborting the atomic block, so that the exception could be thrown with the contents written into it by the atomic block. But if the exception contained pointers to other memory that was allocated or modified by the atomic block, would *those* allocations and modifications also be retained? Determining what additional state should be preserved is not feasible because, for example, the type unsafe nature of C++ precludes determining exactly what data is referred to by pointers, array indices, and so on in the exception object.

Semantic issues aside, abort-on-escape might be overkill in some cases. Consider, for example, using atomic blocks to make a sequential library thread-safe. If the library already provides strong exception safety guarantees then commit-on-escape is more efficient and avoids the semantic issues with abort-on-escape.

The lack of a precise and reasonable answer to the question of what to do with exceptions when aborting makes it difficult to make a compelling case for abort-on-escape semantics, and yet we find commit-on-escape semantics unacceptable, as explained above. The debate in the community has gone around these issues and found no comfortable resolution.

Recently, we have realized that *both "sides" are right* in their objections to the other: some programs have surprising behavior under commit-on-escape semantics, while others have surprising behavior under abort-on-escape semantics. This led us to conclude that *any* situation in which an exception can escape an atomic block without this being clear when examining the atomic block's code is potentially dangerous. Furthermore, different behavior may be appropriate in different circumstances, and only the programmer can hope to determine when the atomic block should be committed, when it should be aborted, and what exception should be thrown if it is aborted.

The proposal outlined in this paper enables programmers to explicitly indicate which types of exception may escape the atomic block and commit it; if an exception of a different type escapes the atomic block, the program is required to terminate, just as is already required in C++ if an exception escapes from `main()` without being caught. It further supports explicitly aborting an atomic block and throwing an exception out of it.

We believe that this approach addresses the main criticisms of each of the commit-on-escape and abort-on-escape approaches. First, a programmer is much less likely to overlook the fact that an atomic block may be committed without completing if it must be explicit that this can happen. And second, as a result of making the programmer explicitly choose the abort-on-escape

approach, it becomes easier to define a set of rules determining when it is safe for an exception to leave the scope of a transaction being aborted, and have programmers obey these rules.

Despite the progress this proposal represents, there are still different opinions on what the *default* behavior should be in case there is no explicit treatment of an exception. Some of us are strongly in favor of making it an error, while others favor a default commit-on-escape semantics. Thus, in some sense, the "big" difference of opinion remains. Interestingly, however, the syntactic proposal outlined in this paper is acceptable to supporters of both positions, and allows programmers to determine whichever behavior is appropriate for their circumstances, rather than having it imposed on them by a language design decision. This allows us to move forward on the issue, to implement the proposed syntax, and to gain experience that will help inform the debate about the eventual decision on default behavior.

We introduce the basic elements of our proposal in Section 2, and discuss remaining unresolved issues in Section 3. We conclude in Section 4.

2. Our proposal

In this section we discuss our proposal in detail. The proposal has the following central principles:

- Programmers should be able to explicitly state which exceptions can escape the scope of an atomic block in order to avoid silent errors when an exception is overlooked;
- the language should provide the programmer with the flexibility to commit *or* abort an atomic block when an exception escapes its scope; and
- the state of an exception thrown out of an aborted atomic block should be preserved; however, it is the programmer's responsibility to ensure that the exception makes sense after the block is aborted

2.1 Exception specifications for atomic blocks

Our proposal builds on C++ *exception specifications* [5] to support explicit declaration of which exceptions may escape an atomic block. A C++ function definition (or declaration) can be augmented with an exception specification, which lists exception types the function is allowed to throw. An attempt to throw any other exception type from a function thus adorned results in a call to `std::unexpected()`, which normally terminates the program. The following example declares that `f1()` may throw an exception of type `int` and `f2()` will not throw any exception:

```
void f1() throw(int);
void f2() throw();
```

As with functions, attaching an exception specification to an atomic block makes it explicit that exceptions may escape the atomic block. The following example illustrates the use of an exception specification to declare that an exception of type X or Y may escape this atomic block:

```
__tm_atomic throw(X, Y) {<stmts>}
```

If an exception of any other type escapes the scope of this atomic block, this will result in a call to `std::unexpected()`. The following example similarly declares that *no* exception will escape this block:

```
__tm_atomic throw() {<stmts>}
```

We further propose syntax to declare that an exception of *any* type may escape this atomic block:

```
__tm_atomic throw(...) {<stmts>}
```

Exception specifications attached to an atomic block alert programmers that the block may not execute in its entirety, hopefully reducing erroneous reasoning such as in the example in Section 1.

2.2 Commit-on-escape

Our concerns surrounding committing an atomic block when an exception escapes it are not about semantics, which are natural and simple, but that programmers can easily overlook the fact that it may happen and therefore incorrectly reason about the semantics of a given atomic block. Therefore, we feel it is sufficient to require explicit acknowledgment that this may occur, so that it is not so easily overlooked. Adding exception specifications to atomic blocks as described above achieves this.

On the other hand, if the atomic block is to be aborted, there *are* semantic issues regarding the exception to be thrown, and different behavior will be desirable in different circumstances, so the language should allow the programmer to determine behavior in this case.

For these reasons, our proposal commits an atomic block when an exception that is listed in the exception specification escapes the atomic block, unless the programmer explicitly specifies that it should abort. So in the following example, any exception thrown by the call to `f1()` (which can only throw exceptions of type `int`) will commit the transaction:

```
__tm_atomic throw(int) {
  f1(); // may throw int per prior declaration
}
```

The programmer can use the following idiom to allow any exception to commit a transaction unless the programmer has explicitly specified that it aborts:

```
__tm_atomic throw(...) {
  f1() // any exception thrown by f1() commits
}
```

2.3 Abort-on-escape

Programmers specify that a given exception should abort the atomic block it escapes as follows:

```
__tm_abort throw x;
```

This *abort-and-throw* syntax combines the `__tm_abort` statement—which is used to explicitly abort an atomic block—with the `throw` statement. It simultaneously specifies that the atomic block should be aborted, and that the given exception should be thrown from the aborted atomic block. In some cases the exception thrown may be one that has been thrown by some code in (or called from) the atomic block, and in other cases it may be more appropriate to construct a different exception based on state observed from within the atomic block before it is aborted.

We believe an exception should only be so "blessed" to escape an atomic block from lexically within that block, again making it more difficult to overlook the possible interaction of exceptions with an atomic block when examining the code for that block. Therefore, an abort-and-throw statement only permits the exception to escape the inner-most atomic block. If the exception is to be thrown through additional enclosing atomic blocks, these blocks must catch and "re-bless" the exception with an abort-and-throw statement.

We note that the following simple idiom can be used if the desired behavior is that *any* exception aborts the atomic block it escapes and propagates to a catch block higher up the stack.

```
__tm_atomic throw(...) {
  try {
    <stmts>
  } catch (...) {
    __tm_abort throw;
  }
}
```

2.4 Exceptions thrown from aborted blocks

We have not yet addressed the semantic issues regarding the exception itself in the case that the atomic block is aborted. If the exception was allocated and/or modified within the atomic block, then it may contain or refer to state that will not be meaningful after the atomic block is aborted. As explained earlier, we do not believe a one-size-fits-all answer to this question is feasible. Instead, we describe a mechanism that gives natural default behavior for many simple cases, but can be overridden by the programmer when appropriate.

The first concern is that aborting the atomic block may destroy the exception object or roll back changes made to it within the atomic block. However, this issue exists already in C++ for exceptions that are allocated on the stack (which is about to be unwound), and the problem is already addressed. Specifically, when an exception is thrown, the C++ runtime typically clones it into a special "exception area" (using the exception's copy constructor).

By preventing the rollback from undoing the copies made in the special area, we ensure that there is no issue for simple exceptions. Programmers can provide copy constructors for more complicated ones that refer to additional state that may be rolled back. Programmers may also opt to replace the thrown exception with a new one, perhaps of a different type, indicating that an exception was thrown from an aborted atomic block.

We note that, while our approach is convenient and flexible, it does not prevent all possible mistakes. In particular, a programmer may overlook the fact that an object whose pointer is copied by a complicated copy constructor is destroyed as a result of a transaction being aborted, which would result in a dangling pointer. Consequently, it is the programmer's responsibility to ensure that the exception object is meaningful after the atomic block is aborted; otherwise unexpected behavior is possible.

Additionally, some subtle issues arise if an exception is rethrown inside an atomic block, particularly related to inheritance. Consider the following example:

```
__tm_atomic throw(x) {
  try {
     foo();
  } catch (x& e) {
     __tm_abort throw; // rethrows exception
  }
}
```

If `foo()` throws an exception that is a subtype of `x` the abort-and-rethrow statement will rethrow that exception after aborting the transaction. In this example, the programmer may think that the re-thrown object is of type `x` and after inspecting the type of `x` (including its copy constructor) conclude that it's safe to abort and rethrow it. In reality, however, the rethrow may throw an object that is a subclass of `x` that is not safe in the presence of abort. One can make this code safer by explicitly naming the thrown exception (e.g., `__tm_abort throw e;`) so that the programmer specifies the exact type of the thrown exception.

3. The debate over default behavior

The syntax described in the previous section leaves the following question unanswered: What behavior is required when an exception escapes an atomic block that is unadorned by an exception specification?

This seemingly minor question captures the essence of the main remaining disagreement amongst the authors and others. The two dominant positions are that the absence of a `throw` clause should be interpreted as

if a `throw()` clause were present, and that it should be interpreted as if a `throw(...)` clause were present.

In other words, some people would like all exceptions that escape such an atomic block to commit the atomic block, and others would like them to cause the program to terminate. We discuss both options below.

3.1 Commit by default

This option is supported by people who say that exception propagation is just a non-local control transfer in some environments, particularly in managed languages [8], so the atomic block should be committed just as it would be if control left it in some other way. This provides an appealing property that adding atomic blocks to a program does not change its single-threaded semantics. Furthermore, it fits naturally with simple proposals for specifying TM semantics, such as the Single Global Lock Atomicity (SGLA) model [6].

Those opposing this option point out, as described in more detail in Section 1, that silently committing an atomic block whenever an exception escapes its scope violates the atomicity property of this block.

3.2 Error by default

Those supporting this option believe that any time an exception escapes from an atomic block without the programmer being explicitly aware of this possibility, there is likely a bug. Silently ignoring the bug will lead to unpredictable behavior and difficulty diagnosing the bug. Therefore, they consider it better to cause a runtime error in this case, which will draw attention to the bug immediately. Furthermore, this approach will encourage programmers to think carefully about the interaction between exceptions and atomic blocks, eliminating errors caused by a default approach behaving unexpectedly.

An argument against this choice is that it is at odds with the choice made for exception specifications for functions, where no specification means any exception can be thrown; this difference may be confusing.

3.3 Resolving the debate

Although significant differences remain between supporters of the two options mentioned above, we consider our proposal to be a valuable contribution towards resolving the issue. It establishes significant common ground, such that implementations can proceed, experience can be gained, and debates on the issues can be made more concrete. In particular, it would be a trivial amount of additional work to support both options, perhaps with compiler switches to select between them, allowing the two proposals to be easily compared.

4. Concluding remarks

Almost all prior proposals for integrating exceptions with atomic blocks favor a default behavior—either commit [8] or abort [3]—when an exception escapes an atomic block, based on opinions about "normal" usage of exceptions. Harris [2] proposes an intermediate solution with commit-on-escape by default, but abort-on-escape for special `AtomicAbortException` exceptions. Like us, Crowl et al. [1] point out that both commit/abort semantics may lead to surprising program behavior in different circumstances, thus compromising the promise of TM to simplify concurrent programming.

We believe we have made significant progress towards a reasonable way to integrate exceptions and atomic blocks, at least in the context of C++. Our approach gives the programmer flexibility to achieve the behavior most appropriate to the given situation, and requires explicit treatment of potentially dangerous cases, making them harder to overlook. We hope that our proposal will encourage useful and interesting discussions—which can provide input for resolving remaining issues—at HotPar and beyond.

Acknowledgments: This proposal has benefitted from our discussions with many people, including but not limited to Steve Clamage, Lawrence Crowl, Robert Geva, Dan Grossman, Tim Harris, Clark Nelson, Tatiana Shpeisman, and Douglas Walls.

References

[1] L. Crowl, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Integrating Transactional Memory into C++. In *TRANSACT*, 2007.

[2] T. Harris. Exceptions and side-effects in atomic blocks. In *CSJP*, 2004.

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

[4] M. Herlihy and E. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.

[5] International Organization for Standardization. *ISO/IEC 14882:2003(E) Programming Languages – C++, Second Edition*. 2003.

[6] V. Menon, S. Balensiefer, T. Shpeisman, A. Adl-Tabatabai, R. Hudson, B. Saha, and A. Welc. Practical weak-atomicity semantics for Java STM. In *SPAA*, 2008.

[7] Y. Ni, A. Welc, A. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R.Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for C/C++. In *OOPSLA*, 2008.

[8] M. Ringenburg and D. Grossman. AtomCaml: First-class atomicity via rollback. In *ICFP*, 2005.