Document Number: N3591
Date: 2013-03-15
Authors: Hans Boehm, HP, hans.boehm@hp.com
Justin Gottschlich, Intel, justin.e.gottschlich@intel.com
Victor Luchangco, Oracle, victor.luchangco@oracle.com
Maged Michael, IBM, maged.michael@acm.org
Mark Moir, Oracle, mark.moir@oracle.com
Torvald Riegel, Red Hat, triegel@redhat.com
Michael Scott, University of Rochester, scott@cs.rochester.edu
Michael Spear, Lehigh University, spear@cse.lehigh.edu
Tatiana Shpeisman, Intel, tatiana.shpeisman@intel.com
Michael Wong, IBM, michaelw@ca.ibm.com
Project: Programming Language C++, SG5, EWG
Reply to: mark.moir@oracle.com
Revision: 1

Summary of Discussions on Explicit Cancellation in Transactional Language Constructs for C++

Introduction

This document summarizes discussions in the SG5 Transactional Memory Study Group on ideas for supporting explicit cancellation of atomic transactions. While we have not reached agreement on whether to support such features or how to do so if we do, we have had some useful discussions that have generated some new ideas and have helped us understand some issues in how we present such features.

Other accompanying documents are:

- Draft Specification of Transactional Language Constructs for C++(v1.1) [1]
- Summary of Progress Since Portland towards Transactional Language Constructs for C++ (N3589) [2]
- Alternative proposal for cancellation and exceptions (N3592) [3]

1. Directions for explicit cancellation and nesting

The ideas discussed in this document are not intended for the next version of the specification, but rather to capture some of the progress made in recent discussions, and update people on them. We do not all agree that features like those discussed in this document are important, and amongst those who agree that features along these lines are important, there is no agreement about what the right way to support them is.

Nonetheless, the discussions since Portland have been very helpful, both in understanding difficulties communicating about some proposed approaches, and in generating new ideas for more flexible and general features to consider in the future. We briefly summarize some of these learning below.

2. Explicit cancellation

The ability to explicitly cancel a transaction has the potential to simplify programming and improve performance in some cases. Essentially, if a programmer is able to say “cancel the effects of this transaction”, then there is no need for him/her to maintain sufficient information while executing the transaction to allow its effects to be “undone” (more accurately, allows its effects to be extended so that the net effect is as if nothing happened).

Furthermore, the sooner the implementation can be informed of the programmer’s intent to cancel the transaction, the sooner it can use optimizations that exploit this knowledge. Consider the following example:

```
void bar() {
  ...
  if (<some condition>)
 <please cancel the transaction>;
}


void foo() {
  SomeType obj;
  ...
  bar();
  ...
}

void main() {
  ...
  __transaction_atomic {
 foo();
  }
  ...
}
```

In this example, the code in `bar()` can decide in some circumstances that it wants to cancel the transaction's effects. By explicitly saying so as soon as this decision is known, various optimizations can be applied. For example, there is no need to execute the destructor for `obj`, because the effects of the constructor will be undone when the transaction is canceled (as will the effects of the destructor if it is executed).

On the other hand, if the decision to cancel the transaction is not known until control returns to the transaction boundary, then these optimizations cannot be applied, and furthermore the programmer must explicitly ensure that control returns to the transaction's boundary, and that the (unnecessary) execution of code between the decision point and the transaction boundary remains safe.

3. Explicit cancellation in previous versions

Two forms of explicit cancellation are included in version 1.1 of the specification [1]. The first is “cancel outer”, which can cancel only the outermost transaction. This was included in an effort to avoid the need to implement closed nesting, but the group has decided it wants closed nesting anyway (see below).

To facilitate static checking to ensure that “cancel outer” is never invoked when not executing within a transaction, previous versions of the spec included various syntactic elements and rules programmers must follow. Pretty much everyone agrees in hindsight that this was too much overhead on the language for the benefit it gives. The main reason is that, if a buggy program attempts to cancel a transaction when it is not in one, this can be immediately identified and reported as a bug in a clear and predictable way. Thus, it differs from other errors that we still want to avoid via static checking, such as errors that might allow silent atomicity violations, whose consequences may be unpredictable and may occur long after the error actually occurs.

The other form of explicit cancellation included in previous versions is “lexically nested cancel”, which is limited to cancel a transaction within which it is lexically nested. Thus, it has the more or less the same disadvantages as discussed above regarding the need for control flow to return to the lexical scope of the to-be-canceled transaction before the decision to cancel can be expressed.

It was a deliberate decision to include these two forms of explicit cancellation in previous versions in order to learn the relative advantages and disadvantages. Nonetheless, the nonuniformity of the two forms, the lack of flexibility, and the other disadvantages mentioned above left previous versions of the specification without a compelling and coherent position on explicit cancellation.

4. Possible future directions for explicit cancellation

Our recent discussions on these issues have given rise to a set of ideas that has the potential to address all of the disadvantages discussed above. Nonetheless, we have not achieved agreement on whether such features are important, nor is there agreement on how the features should be specified even amongst those who agree on their importance.

Here, we briefly summarize some of the learning from these discussions. We have decided to sidestep these issues for the next version of the specification by not supporting any explicit cancellation besides throwing an exception from a transaction labeled as `[[cancel_on_escape]]` (N3589).

Before we discuss the ideas arising from these discussions, we first make some observations based on the above skeletal example. The “lexically nested cancel” proposed previously cannot support use cases such as that shown in the example, because the decision to cancel is not expressed in the lexical scope of the transaction to be canceled.

The “cancel outer” feature proposed previously can support this case, but in an inflexible way. This is due in part to the risk that the transaction shown in `main` above could subsequently be enclosed in another transaction, and would therefore no longer be the outermost transaction. The surprises this could cause were the main motivation for allowing cancel-outer only within a transaction explicitly annotated as being an “outer” transaction; this annotation allowed the compiler to prevent an outer transaction from being enclosed in another transaction.

As mentioned, we have decided to support closed nesting (meaning that it is possible to cancel the effects of a transaction that is nested within one or more other transactions, without canceling the effects of the enclosing transactions). This raises the possibility of being able to cancel a specified number of levels of transactions, and it turns out that this gives rise to a much cleaner and more uniform set of features that would support cancellation of the immediately enclosing transaction, of the outermost transaction, or of any number of enclosing transactions within the outermost transaction, using a single, unified set of features.

The key to the uniformity, generality, and flexibility of the approach we describe next is the ability to **identify** the transaction to be canceled in some way that is robust to changes in other code. For example, if the cancel in the example above were able to explicitly identify the transaction to be canceled, then it would not matter if either that transaction were subsequently enclosed in another one, or if intermediate code---such as in `foo()`---in the example, were changed so that the number of transactions between the to-be-canceled transaction and the cancel statement changes. There are several ways this could be achieved. The one we have discussed most concretely is briefly summarized next.

First, we allow the cancel statement to specify a number of levels of transactions to cancel. If this can be expressed only as a constant, it does not help with the problems mentioned above. On the other hand, if we provide a library function, such as:

```
 std::transaction_nesting_level()
```

that returns the number of nested transactions at the point at which it is called, then this can be used to identify a) the nesting level of the transaction to be canceled, and b) the nesting level of the cancellation statement. The difference between these determines the number of levels that need to be canceled in order to cancel the desired transaction. This also provides the ability to express “cancel-outer” by cancelling the number of levels indicated by `std::transaction_nesting_level()`.

In at least some cases, it should be possible to optimize the “cancel-outer” case using hardware transactional memory (HTM).

Our discussion of these ideas has mostly been in the context of an attempt to extend the one case in which we know we **need** to be able to cancel transactions (i.e., when an exception escapes a transaction; see N3589) to more general and explicit cancellation. Because of this desire, the way we chose to express cancel was by throwing a special kind of exception that would have special behavior (essentially, cancel the transaction so that it is as if it did nothing except produce an exception).

However, it turns out that this causes too much confusion, because it causes some people to think about behavior of “normal” exceptions, and therefore get worried about how the cancellation exception would interact with intermediate exception handling code. The conversation became bogged down by this confusion and it became clear that it had probably been a mistake to try to extend the “cancel via exception” metaphor.

The basic ideas and advantages do not change if we express “cancel some number of levels of transactions” in different ways that do not evoke thoughts of normal exception processing. For example, a new keyword could be used, or a library call could be used.

A related question is how the code surrounding the canceled exception learns that the transaction was canceled and what information it receives to determine why it was canceled and/or to guide its decision about what to do next. We have discussed this in some detail in the context of the overextended exception metaphor, and there have been other ideas floated---but not yet fleshed out in detail---regarding how such information is generated and communicated out of the canceled transaction.

Several of us remain convinced that explicit cancellation will be a valuable tool, especially if it is fast enough to use for common cases, not just error or unexpected conditions. Others are less convinced this kind of functionality is useful in enough cases to justify inclusion of the features.

This summary merely attempts to capture the essence of recent discussions, and this topic is deferred for possible further discussion for the time being.

Conclusion

Discussions on flexible explicit cancellation features that can be used anywhere within the dynamic scope of an atomic transaction have led to new ideas that aim to simplify and unify the disparate forms of explicit cancellation included in previous versions of the specification. Nonetheless, the SG5 group has not reached agreement that such features are needed, and furthermore, even those that agree that they should be supported have not reached consensus on the particular way in which it is expressed. At least some members of the group hope these discussions will be revisited later, but there are no plans for now to include these features in the next versions of the specification.

Acknowledgement:

This proposal is the culmination of work by at least the following individuals who contributed to the "Draft Specification of Transactional Language Constructs for C++." [1]

Ali-Reza Adl-Tabatabai (Intel), Kit Barton (IBM), Hans Boehm (HP), Calin Cascaval (IBM), Steve Clamage (Oracle), Robert Geva (Intel), Justin Gottschlich (Intel), Richard Henderson (Red Hat), Victor Luchangco (Oracle), Virendra Marathe (Oracle), Maged Michael (IBM), Mark Moir (Oracle), Ravi Narayanaswamy (Intel), Clark Nelson (Intel), Yang Ni (Intel), Daniel Nussbaum (Oracle), Torvald Riegel (Red Hat), Tatiana Shpeisman (Intel), Raul Silvera (IBM), Xinmin Tian (Intel), Douglas Walls (Oracle), Adam Welc (Intel), Michael Wong (IBM), Peng Wu (IBM)

We also wish to acknowledge contributions from David Abraham, Jens Maurer, Lawrence Crowl, Paul McKenney, Nevin Liber, Daniel Krügler, Niall Douglas and any others who joined the discussion on the reflector or at meetings that we have left out unintentionally.

Reference

[1] Draft Specification of Transactional Language Constructs for C++(v1.1) :
https://sites.google.com/site/tmforcplusplus/
[2] WG21 N3589 : Summary of Progress Since Portland towards Transactional Language Constructs for C++
[3] WG21 N3592 : Alternative proposal for cancellation and exceptions