Document Number: N3589
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++, EWG
Reply to: victor.luchangco@oracle.com
Revision: 1

Summary of Progress Since Portland towards Transactional Language Constructs for C++

Introduction

This document summarizes progress that has been made by the SG5 Transactional Memory Working Group since the Portland meeting and is a follow-on to N3341 Transactional Language Constructs for C++ [4] presented in Kona in 2012. Since then, Transactional Memory was presented in May 2012 in a special advanced Parallel Summit hosted by Microsoft at which point, it was agreed by all present to form SG5 Transactional Memory Subgroup for the purpose of future C++ standardization of Transactional Memory. This subgroup has been meeting by telecon twice monthly making steady progress, held a 3 days SG5 face-2-face meeting at Portland (mostly in Vietnamese restaurants) and presented status report at the full plenary.

We restrict attention in this document to issues that we have agreed upon for the next version of the specification, which we are preparing. Other accompanying documents are:

- Draft Specification of Transactional Language Constructs for C++(v1.1) [1]
- Summary of discussions on explicit cancellation (N3591) [2]
- Alternative proposal for cancellation and exceptions (N3592) [3]

1. Necessity of cancellability

In hallway (and Vietnamese restaurant!) conversations in Portland, there was a lot of discussion about whether trivial implementations that simply use a global lock to implement transactions and use Hardware Transactional Memory (HTM) to address the resulting scalability bottleneck using lock elision would be viable. The conclusion is that this is not the case (for a summary, see the email by Mark Moir titled “Atomic transactions should be cancelable” from 27 Oct 2012 on the reflector). We have agreed on this point, even though some implementors may use the above-described simple implementation for limited support of a subset of the desired features in the short term.

2. Relaxing `transaction_safe` requirements

In Portland, we reported that we have received feedback to the effect that the need to apply the `[[transaction_safe]]` attribute[1] to functions intended for use in atomic transactions, and to make all declarations and definitions of such functions consistently do so, was too much burden on programmers. In particular, they found it frustrating that they had to make so much effort before they could even compile and run their program during development and testing.

To address this issue, we have decided to move to a "safe by default" model, in which a function is **assumed** to be safe for use in atomic transactions unless i) there is unsafe code in the function, or ii) the function is explicitly labeled `[[transaction_unsafe]]`.

While the compiler can determine whether there is unsafe code directly in a given function, it cannot always make this determination for all functions called directly or indirectly from it. This is the reason that previous version of the specification [1] have insisted that the `[[transaction_safe]]` attribute be consistently applied to all declarations and definitions of a function intended for use in atomic transactions: it allows the compiler to statically ensure that all functions called within an atomic transaction do not contain unsafe code, even if some of those functions are in different compilation units.

In the "safe by default" model, the compiler **assumes** that any function that is called---directly or indirectly---from a given function does not contain any unsafe code unless there is an indication to the contrary. The compiler and linker cooperate to ensure that all assumptions are validated before a program can be run. In particular, they ensure that no function that can be called from within an atomic transaction contains any unsafe code. (We explain below how this restriction can be relaxed via the `__forbidden_in_atomic` construct.)

This approach makes it unnecessary for the compiler to be told exactly which functions are thought to be safe for use in atomic transactions, and thus eliminates the need for consistent labeling in every declaration and definition of every function. Consider this simple example:

file1.cpp

```
void foo() {
   <safe stuff only>
   bar();
}

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

file2.cpp

[1] We recognize that attributes may not be the best syntax for the proposed features. However, to avoid confusion, we retain this syntax in our current discussions.

```
void bar() {
  <unsafe stuff here>
}
```

Although the transaction can call a function (`bar`) that contains unsafe code, the compiler does not issue an error when compiling either of these files, because there is no way for it to know when compiling file1.cpp that `bar()` is not transaction-safe (so it assumes bar is transaction-safe), and there is no way for it to know when compiling file2.cpp that `bar()` is intended for use in a transaction.

However, the linker will have all information available to it to determine that `foo` is called inside a transaction, `foo` calls bar, and `bar` is not transaction-safe. Thus, the linker can report an error in this case, and thus we ensure that the program will not run if unsafe code can be executed within an atomic transaction.

While this approach imposes much less burden on programmers, it does have some potential downsides: inflexibility, code bloat, increased compilation time, and the use of functions in atomic transactions that are not intended for that purpose (and may not be suitable for it in the future).

The inflexibility mentioned above arises from the restriction that a function intended for use in an atomic transaction cannot include any unsafe code. As described so far, this restriction applies even to functions that would never actually execute the unsafe code when called from within an atomic transaction. To address this issue, we allow programmers to explicitly indicate that a given block of code is not intended to be executed within an atomic transaction; this is indicated using the `__forbidden_in_atomic` keyword to demarcate such a block of code. A function that contains no unsafe code **except** within such a block is deemed to be safe for use in atomic transactions. If the belief expressed by the programmer is incorrect, and in fact the program attempts to execute such a block of code within an atomic transaction, a runtime error will be displayed and the program will terminate.

The remaining potential issues all stem from the possibility that the compiler will assume that some functions are intended for use in atomic transactions when in fact this is not the case. If such functions are in fact not used in atomic transactions, then link-time optimizations may be able to automatically address code bloat. But this will not address the other two potential issues: the compiler will still spend time unnecessarily preparing the functions for use in transactions, and if someone uses a function (say a library function) in an atomic transaction when the implementor of that function does not intend this to be supported, then future changes to that function might make it no longer usable in that context.

For all of these reasons, we still plan to support the `[[transaction_safe]]` and `[[transaction_unsafe]]` attributes. This allows programmers to mark as transaction-unsafe any functions not intended for use in atomic transactions, which will allow them to address all of the issues mentioned above. Note that they can do so incrementally, after they are already running and testing their code.

Furthermore, we will no longer require that all function definitions and declarations consistently use these annotations. Instead, we will simply require that there are no **inconsistent** annotations: if a function is annotated as transaction-safe in one definition or declaration, it cannot be declared as transaction-unsafe in another, and vice versa. Note that violations of this rule might not be reported at compile time, but will be reported at link time at the latest.

The advantages of using these annotations, even though they are optional, include:

- `[[transaction_unsafe]]`
    - Avoiding unnecessary compilation time and code bloat for functions not intended for use in atomic transactions.
    - Avoiding unintended use of functions in atomic transactions, preserving the right to use unsafe code in future implementations without breaking client code.

- `[[transaction_safe]]`
    - Allowing accidental use of unsafe code in a function intended for use in atomic transactions to be caught at compile time, rather than only at link time when someone tries to link a program that actually calls the broken function. (Note that, if a programmer (eventually) adheres to the stricter rules imposed in the previous version of the specification[1], then errors due to accidental use of unsafe code in functions intended for use in atomic transactions will always be caught at compile time at the latest, just as with the previous version. The difference is that this can be done incrementally, after the program is already running, rather than requiring these rules to be obeyed before running for the first time.)
    - Communicating to users the intent that the function is safe for use in atomic transactions, allowing them to be more confident that this will continue to be true.

Finally, we note that version 1.1 of the specification [1] allowed for some functions to be "implicitly declared safe". Considering functions to be declared safe by default is closely enough related that the safe-by-default direction might be considered to just be a broadening of the scope of "implicitly declared safe". But in practice there are some important differences, as summarized above. In particular, with the new approach, the compiler does not need to be able to see all code that might be called from a function in order to allow its use in atomic transactions.

3. SEMANTICS OF `__transaction_relaxed`

There has been a lot of confusion over the semantics and purpose of relaxed transactions, and how they compare to atomic transactions. Even within the drafting group, there have been different opinions and significant confusion in discussions over it.

We have made significant progress on this front, in that we have agreed that the semantics of relaxed transactions can be stated in terms of features that people already understand, namely locks. Briefly, the semantics is the same as if the body of the relaxed transaction were in the critical section of a specially designated lock and atomic transactions do not take effect while the lock is held. (Of course, this is the semantics, and does **not** dictate that it must actually be implemented literally this way.)

The clarification of the semantics of the `__transaction_relaxed` keyword has also enabled us to see some potential ways of specifying the same semantics in a way that may allow for much more flexibility and better performance and scalability. However, while we have some ideas in this direction, we have not fleshed them out in detail or decided on an approach, and furthermore we don't yet have the problem that would necessitate doing so (such as people complaining of scalability issues due to the basic use of the `__transaction_relaxed` keyword).

These discussions have been very helpful in understanding the semantics and purposes of the `__transaction_relaxed` keyword, and have helped some of us understand each other's positions better. Nonetheless, we have decided to postpone for now discussions of whether these insights should result in changes in spelling or syntax for the `__transaction_relaxed` keyword.

We have therefore agreed for our next version to keep the keyword and its spelling, and to elucidate the semantics so that we'll be in a better position later to have a productive discussion about how the syntax and/or spelling should eventually look.

4. Minimal exceptions proposal

We have spent a lot of time discussing nesting, exceptions, cancellation, and the interplay between these issues. This has resulted in some ideas that some of us think are powerful and worth pursuing, while others are not so convinced. Some of these ideas are discussed in separate documents (N3591 and N3592). However, having not reached agreement about these ideas, we have decided that our next draft will include:

- A minimal exception proposal

  This proposal, summarized below, allows for transaction cancellation **only** in the case that an exception escapes a transaction that is explicitly annotated as one that will be canceled if an exception escapes it.

- Support for closed nesting

  Previous attempts to avoid the implementation complexity of "closed nesting" (the ability to cancel the effects of a nested transaction but commit the remaining effects of the transaction within which it is nested) led to semantics that some people didn't like, and syntactic overhead that nobody likes (anymore). Therefore, we have decided to bite the bullet and specify closed nesting semantics.

- No explicit support for "cancellation at a distance" for now

  This means that, if some code being executed within a transaction decides that the transaction's effects should be canceled, it will be necessary to communicate this back to the boundary of that transaction using conventional means (e.g., exception propagation or return codes). The consequence of this decision is that it imposes some additional burden on programmers and disables some optimizations that are considered important by some of us. On the positive side, it allows us to make progress on other important aspects of the specification while leaving open the possibility to revisit such features in the future. Some of these ideas are briefly discussed in the next section, but we have agreed not to hold up the next version of the specification in order to resolve these issues.

The minimal exception proposal requires the exception handling behavior of every atomic transaction to be stated explicitly using one of the following three attributes: `[[noexcept]]`, `[[commit_on_escape]]`, `[[cancel_on_escape]]`.

The reasons to require all cases to be stated explicitly are that there are potential surprises for any choice of default, and that there is significant diversity of opinion about which of these surprises are most important to avoid, and thus there is no agreement on what the default should be if there is one. By requiring all handling to be indicated explicitly for now, we make all of the surprises less likely, while leaving open the possibility to agree on a default behavior without breaking existing programs in the future, once there is more experience with these features.

Although the minimal exception proposal defers a number of tricky issues, it still requires us to come up with a reasonable medium-term answer to the question of what exceptions can escape from canceled transactions, and how those that cannot are handled. We have decided for the next version of the specification that we will also consider an alternative proposal on cancellation and exception [3].

Conclusion

The transition from the original drafting group to a formal Study Group (SG5) has brought in a number of new people, new perspectives, and new expertise. This has greatly enhanced our discussions, and led to significant progress, both in agreeing on how to handle some issues and in elucidating other issues and stimulating new ideas, even if we have not yet converted these to agreement. The group will continue work on completing drafting the next version 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 N3591 : Summary of discussions on explicit cancellation

[3] WG21 N3592 : Alternative proposal for cancellation and exceptions

[4] WG21 N3341 : Transactional Language Constructs for C++