Document Number: N4000
Date: 2014-05-23
Authors: Michael Wong, michaelw@ca.ibm.com
Jens Maurer, jens.maurer@gmx.net
Michael Spear, spear@cse.lehigh.edu
Justin Gottschlich, justin.gottschlich@intel.com
Torvald Riegel, triegel@redhat.com
with other members of the transactional memory study group (SG5), including:
Hans Boehm, hans.boehm@hp.com
Victor Luchangco, victor.luchangco@oracle.com
Maged Michael, magedm@us.ibm.com
Mark Moir, mark.moir@oracle.com
Michael Scott, scott@cs.rochester.edu
Tatiana Shpeisman, tatiana.shpeisman@intel.com
Project: Programming Language C++, LWG, SG5 Transactional Memory
Reply to: Michael Wong michaelw@ca.ibm.com (Chair of SG5)
Revision: 1

Standard wording for a Transaction-safe C++ Standard Library std::list

Abstract

This paper documents our effort to transactionalize a C++ Standard Template Library (STL) container to demonstrate the feasibility of the transactional language constructs proposed by *Study Group 5 (SG5): Transactional Memory*. We began this study with `std::list` and made it transaction-safe using the transactional memory support in GCC 4.9. The changes were minimal and were generally restricted to the addition of `transaction_safe` keyword to a few interfaces such as allocate, deallocate, and swap functions. The rest of the changes were added to internal helper functions. Some of the issues that we considered were the constant time complexity of `std::list.size()` and friends, and its `const noexcept` nature. This experience shows that the safety of STL containers must not be specified directly, but instead should be inherited from the type with which the container is instantiated. For our future work, we plan to expand this effort to other STL containers as well as converting the clang/llvm C++ library.

Changes from previous versions

N4000 (this paper): present updates and summary from LEWG and propose wording for TM TS for LWG

N3862: presented to LEWG for review of transactionalizing std::list. Urged to continue work [9] [10] and approved 8/3/2/0/0 after EWG approved N3859 16/6/1/0/0. Formal motion approved as a NP for a TM TS.

1. Introduction

This paper follows our initial presentation to LEWG on N3862. It updates and summarize the result of discussion from LEWG to enable transactional memory (TM) support in the C++ Standard Library starting with a single container, based on the syntax as described in N3919 [ ], N3859[1], a follow-on paper to N3718[2]. N3859 describes the syntax and semantics of the proposed TS for SG5 Transactional memory, while this paper (Nxxxx) shows the wording on how these extensions can be applied to STL.

One of the key feedback items from the September 2013, Chicago meeting was that in order to facilitate a seamless introduction of transactional memory to C++, we need to provide, at least a transaction-safe C++ Standard Library along with Transactional Memory syntax and semantics from SG5. This support should enable users to use transactional constructs in the first TS delivery of SG5, without requiring users to invent their own techniques to use Standard Library containers with Transactional memory. This was considered a key delivery as part of SG5.

In this paper, we will describe our efforts at making `std::list` transaction-safe. The results show it is readily implementable. We used GNU 4.9's implementation of transactional memory syntax as applied the GNU C++ Standard Library of `std::list` which has been converted to C++11[3]. GNU's implementation is based on N3725, the original Draft Specification of Transactional Language Constructs for C++ Version 1.1 that was published in February 2012. Core Standard wording will be reviewed in a separate paper.

2. Summary of LEWG presentation N3862[10]

In the process of transactionalizing `std::list`, we considered several issues with LEWG.

1. The first is that list member function `size()` is required to take constant time in C++11. Since every list mutation thus needs to modify a counter, it implies the potential for many aborts. Our group considered this and felt that we are essentially taking a non-scalable abstraction (linked list) and unreasonably expecting the use of transactional memory to force it to scale. A more realistic outcome is that some transactional data structures will use `std::list` internally (a specific analogy to STAPL[6] was made).

   Nevertheless, nonscalable transactional data structures can serve useful purposes for building scalable ones. For example, a transactional hash table whose buckets are implemented using std::list would achieve scalability in the common case in which hashing is effective, because individual buckets would not normally experience heavy contention. However, if such a hash table were to include a size method with similar requirements for constant-time execution, the same problem experienced with list nonscalability would apply to the hash table.

   In general, building scalable data structures requires not only effective programming support for concurrency, but also careful design of interfaces so that they do not preclude scalability. For example, omitting the size method for the putative hash table mentioned above would

easily avoid the problem. From there, it is useful to consider what functionality could be added that would be useful without precluding scalability. Possibilities include weaker semantic
guarantees for a size "estimate" method, weaker requirements for execution time, etc.

The comment from LEWG [9] was that they recognized a mistake was made in making size() constant time in C++11. This is not as ideal as before when size() was order linear in C++98 for TM. If you only touch a node, it is better for TM because you touch fewer cache lines and offers more scalability. In the end, all agreed that no action resulted from this issue and that we accept the size() member function is just not as fast as others.

2. The second is a further discussion of const noexcept. Specifically, imagine the following scenario:

   Let a long-running transaction T call `my_vector->size()`. Assume that the calling transaction has never accessed `my_vector` before. In most any STM implementation, T will need to internally allocate some data to be able to log the read, in case of an abort. However, the system can run out of memory when trying to do this. If so, the transaction cannot throw `bad_alloc`, because it is in the middle of a `const noexcept` function. So the transaction must abort and restart in a more pessimistic mode, so that it can avoid logging.

   The reason this is interesting is because it seems that const noexcept functions seem to have "progress guarantees", which, in turn, don't compose with the "progress guarantees" that people in the distributed computing research community use. The members of SG5 do not yet have a conclusive position on this issue.

   The general comment from LEWG [9] was that clause [res.pn.data.races] gives implementations permission to protect internal statics with locks. Jens Maurer pointed out this also lets them protect it with transactions. This section also requires const methods to be thread-safe.

   Constexpr functions were also brought up as a possible corollary to the const noexcept guarantee issue but most felt that it is thread friendly.

   Another possible issue discussed was user-defined comparison operators may not be transaction-safe as it is similar to the noexcept issue in that you don’t want noexcept function to call user code.

3. The third issue is about annotations for transaction safety. This experience shows in a very clear manner that the safety of STL containers must not be specified, but instead inherited from the type with which the container is instantiated. Put another way, if `std::list` is instantiated with a class whose constructor and assignment operators perform an unsafe operation (e.g., I/O), then the compiler should accept the instantiation, but forbid calling (most of) its methods from within a transaction. If `std::list` is instantiated with a primitive type, or a class lacking such unsafe code, then the compiler should not prohibit the

transactional use of any method of the list. We have shown this behavior to be achievable in GCC, with only minor modifications to the existing support for transactions.

4. We also discussed that calling instruction-level synchronization primitives in standard containers is not compatible with the current proposal of TM for atomic blocks. Atomic blocks cannot contain these synchronization primitives, but Synchronized blocks can. Jonathan Wakely [9] indicate that GCC can use drop in replacement to work around this.

5. Finally, Eric Niebler[9] brought on the concern as to what if the user keep exporting template for user-defined types. This might require the need to annotate everything. This lead to the concern as to what can happen if you were to declare a whole class template extern. Although we are uncertain if this is a real issue, we continue to work with the original comments to make progress.

6. Changes to the `std::list` Implementation

We started our investigation with std::list. Our preliminary work suggests that making STL transaction-safe is not an insurmountable task. The impact seems to be minimal. A publicly available repository that stores a fully transaction-safe `std::list` is available at https://github.com/mfs409/tm_stl

For reference, GCC 4.9 is under active development, and we were working with trunk version 206059. In GCC 4.9, transactional memory support is enabled via the `–fgnu-tm` flag. When this flag is enabled, within each compilation unit the compiler will infer the transaction-safety of all functions whose bodies are visible[4]. For calls from a `__transaction_atomic` block to functions whose bodies are not visible, the compiler requires that those functions are annotated as `transaction_safe`. If the functions were not compiled with `–fgnu-tm`, or if they were not, in fact, `transaction_safe`, then linking will fail.

To ensure complete coverage, we manually instrumented every method of the `std::list` container. This was done to ensure that every method was called from a transactional context. We then constructed a program that called every method of `std::list` from within some transaction. In total, there were six instances in which the compiler could not infer transaction safety, and one instance in which the compiler is not yet up-to-date with the latest updates in N3859[1].

The following changes were made, most of them to internal helper functions of GNU:

1. In the file `include/bits/functexcept.h`, the `noreturn` function `throw_bad_alloc` needed to be marked `transaction_safe`. This function, whose body essentially consists of a `throw` statement, is called by member function `pointer allocate(size_type, allocator<void>::const_pointer hint = 0)` in the default allocator of 20.7.9. We also recommend that the default

allocator's methods be specified as `transaction_safe`, to ensure that the default allocator can always be used with STL containers.

2. In the file `include/bits/stl_list.h`, there is a call to `__builtin_abort` that is not safe. SG5 has discussed the need to support `assert()` and `abort()`, and has concluded that since neither calls `atexit()` functions, both can be `transaction_safe`. The GCC implementation does not yet reflect this change, but through a small kludge in our program code we were able to coax the compiler into accepting the call to `__builtin_abort`.

3. In the file `include/bits/stl_list.h`, the `_List_node_base` struct has five methods which are implemented in a .cc file that is compiled separately. Consequently, GCC is unable to infer the safety of these five methods. For our study, we annotated the methods as `transaction_safe`. Should such an approach complicate the process of bootstrapping the compiler (i.e., due to the need to support transactions when building fundamental data structures), these five methods could be moved into a header file, at the cost of possibly increasing the compiler generated code size.

7. Potential Changes to Draft N3797 [7], the 2013-10-13 Working Draft

Standard library

Drafting note: The following guidelines were employed for transaction-safety requirements in the standard library, roughly oriented on the guidelines for `constexpr` and `noexcept`:

- If a function can unconditionally satisfy its contract without invoking user-defined code, it is declared `transaction_safe`. Functions declared `noexcept` strongly hint in that direction. Example: `size()` member function of containers.
- If a function is expected to call potentially user-defined code, that function is specified in prose to have transaction-safe linkage under the condition that all of the potentially invoked user-defined functions are transaction-safe. Example: A copy constructor of a container is only transaction-safe if all required functions of the Allocator are transaction-safe and if all required functions for the contained type T are transaction-safe, where "required function" is defined by the respective requirements tables.
- There is no code-level specification of conditional transaction-safety. When simply calling such functions, there is no issue, because the safe-by-default approach (specified as transaction-safe linkage) provides a transaction-enabled variant of a function whenever possible. Only when the address of a function is taken (e.g. `&std::vector<Foo>::push_back`) will the resulting pointer to (member) function type not be transaction-safe, even though the function actually called might be.

1.Add the keyword **transaction_safe** to 20.7.9 The default allocator [default.allocator]

```
namespace std {
  template <class T> class allocator;

  // specialize for void:
  template <> class allocator<void> {
  public:
    typedef void* pointer;
    typedef const void* const_pointer;
    // reference-to-void members are impossible.
    typedef void value_type;
    template <class U> struct rebind { typedef allocator<U> other; };
  };
  template <class T> class allocator {
  public:
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef const T* const_pointer;
    typedef T& reference;
    typedef const T& const_reference;
    typedef T value_type;
    template <class U> struct rebind { typedef allocator<U> other; };
    typedef true_type propagate_on_container_move_assignment;

    allocator() noexcept;
    allocator(const allocator&) noexcept;
    template <class U> allocator(const allocator<U>&) noexcept;
  ~allocator();

    pointer address(reference x) const noexcept;
    const_pointer address(const_reference x) const noexcept;

    pointer allocate(
      size_type, allocator<void>::const_pointer hint = 0) transaction_safe;
    void deallocate(pointer p, size_type n); transaction_safe;
    size_type max_size() const noexcept;

    template<class U, class... Args>
    void construct(U* p, Args&&... args);
    template <class U>
    void destroy(U* p);
  };
}
```

The following additional wording was taken from:
http://jmaurer.awardspace.info/wg21/tmspec.html
2.Change in 17.5.1.4 [structure.specifications] paragraph 3:

- ...
- *Synchronization:* the synchronization operations (1.10) applicable to the function
- ***Transactions:* the transaction-related properties of the function, in particular whether the function is transaction-safe**
- ...

3.Add a new section in 17.6.5 [conforming]:

17.6.5.16 [lib.txsafe] Transaction safety

For a function unconditionally specified to have transaction-safe linkage, an implementation may declare the function `transaction_safe`. Other functions shall not be declared `transaction_safe`.

4.Change in 18.5 [support.start.term] paragraph 4:
The function `abort()` has additional behavior in this International Standard:

- The program is terminated without executing destructors for objects of automatic, thread, or static storage duration and without calling functions passed to `atexit`() (3.6.3).
- **The `abort()` function is declared transaction-safe.**

Add to 18.6.1 [new.delete] paragraph 1:
... **The library versions of the global allocation and deallocation functions are declared transaction-safe.**
Add a first paragraph to section 18.6.2 [alloc.errors]:
The classes `bad_alloc`, `bad_array_length`, and `bad_array_new_length` are transaction-safe.

For each declaration of a member function in 18.6.2.1 [bad.alloc], 18.6.2.2 [bad.array.length], and 18.6.2.3 [new.badlength], add `transaction_safe`.

5.Change in 18.7.2 [bad.cast]:
The class `bad_cast` defines the type of objects thrown as exceptions by the implementation to report the execution of an invalid dynamic-cast expression (5.2.7 [expr.dynamic.cast]). **The class supports transaction cancellation.**

For each declaration of a member function in 18.7.2 [bad.cast], add `transaction_safe`.

6.Change in 18.7.3 [bad.typeid]:
The class `bad_typeid` defines the type of objects thrown as exceptions by the implementation to report a null pointer in a typeid expression (5.2.8 [expr.typeid]). **The class supports transaction cancellation.**

For each declaration of a member function in 18.7.3 [bad.typeid], add `transaction_safe`.

7.Change in 18.8.2 [bad.exception]:
The class `bad_exception` defines the type of objects thrown as described in ~~(15.5.2 [except.unexpected]).~~ **15.5.2 [except.unexpected]. The class supports transaction cancellation.**

8.Change in 18.10 [support.runtime] paragraph 4:
The function signature `longjmp(jmp_buf jbuf, int val)` has more restricted behavior in this International Standard. A `setjmp`/`longjmp` call pair has undefined behavior if replacing the

`setjmp` and `longjmp` by `catch` and `throw` would invoke any non-trivial destructors for any automatic objects**, or would transfer out of a synchronized block (6.x [stmt.sync]) or transaction block (6.x [stmt.tx])**.

9.Change in 19.2 [std.exceptions] paragraph 3:

... These exceptions are related by inheritance. **The exception classes support transaction cancellation.**

For each declaration of a constructor taking a pointer to const char in 19.2.x, add `transaction_safe`.

10.In 20.7.9.1 [allocator.members], add "`transaction_safe`" to the declarations of the following member functions: address (twice), allocate, deallocate, max_size.

11.Change in 20.7.9.1 [allocator.members] paragraphs 12 and 13:

```
template <class U, class... Args>
  void construct(U* p, Args&&... args);
```

Effects: `::new((void *)p) U(std::forward(args)...)`

***Transactions:* Has transaction-safe linkage if the invoked constructor of U has transaction-safe linkage.**

```
template <class U>
  void destroy(U* p);
```

Effects: `p->~U()`

***Transactions:* Has transaction-safe linkage if the destructor of U has transaction-safe linkage.**

12. Add after 20.8.13 [c.malloc] paragraph 2:

The contents are the same as the Standard C library header <stdlib.h>, with the following changes:

The functions are declared transaction-safe.

[Drafting note: This covers calloc, malloc, free, and realloc.]

13. Change in 20.8.13 [c.malloc] paragraph 7:

The contents are the same as the Standard C library header <string.h>, with the change to `memchr()` specified in 21.8 [c.strings]. **The functions are declared transaction-safe.**

[Drafting note: This covers memchr, memcmp, memcpy, memmove, and memset.]

14. Add after 21.1 [strings.general] paragraph 1:

All functions in this Clause have transaction-safe linkage if the required operations on the supplied allocator (17.6.3.5 allocator.requirements) and character traits (21.2.1 char.traits.require) have transaction-safe linkage.

15. In 21.4.3 [string.iterators], 21.4.4 [string.capacity], 21.4.5 [string.access] add "`transaction_safe`" to the declarations of all member functions.

Add after 26.8 [c.math] paragraph 4:
The contents of these headers are the same as the Standard C library headers <math.h> and <stdlib.h> respectively, with the following changes:

The functions from <stdlib.h> are declared transaction-safe.

[Drafting note: This covers abs, ldiv, rand, div, llabs, srand, labs, and lldiv.]

8. Conclusion and Recommendation

This effort demonstrates the feasibility of making an STL container `transaction_safe`. The authors will continue this effort, to analyze additional containers and develop a comprehensive set of recommended changes to the specification. Thus far, we are pleased to observe that changes are minimal and relatively benign.

We will continue to make additional containers transaction-safe as well as work on equivalent clang-llvm C++ library changes. We plan to move onto `std::vector` next, as well as taking into account any particular strategy or preferred containers that need to be made safe as feedback from the committee. One possible list is:

C++98/03 and C++11

```
std::string
std::vector
std::set
std::multiset
std::map
std::multimap
std::list
std::stack
std::deque
```

New for C++11:

```
std::array
std::forward_list
std::unordered_set
std::unordered_multiset
std::unordered_map
std::unordered_multimap
```

9. Acknowledgements

We wish to acknowledge and thank committee members and others who have given us valuable feedbacks.

10. Reference

[1]N3859, Transactional Memory Support for C++, 2014-01-20, EWG, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3859.pdf
[2]N3718, Transactional Memory Support for C++, 2013-08-30, SG5, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3718.pdf
[3] http://gcc.gnu.org/onlinedocs/libstdc++/manual/status.html#status.iso.200x
[4] N3861: Meeting Minutes of SG5. 2014-01-20. Discussion with T. Riegel during SG5 meeting dated Jan 6
[5] "Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached", by Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. 19th International Conference on Architectural Support for Programming Languages and Operating Systems, Salt Lake City, UT. March 2014.
[6] STAPL: https://parasol.tamu.edu/stapl/
[7] N3797: Working Draft, Standard for Programming Language C++, 2013-10-13, Stefanus Du Toit, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf
[8] N3919, Transactional Memory Support for C++, 2014-02-14, EWG paper, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3919.pdf
[9] LEWG Meeting notes Feb 12, 2014https://groups.google.com/a/isocpp.org/forum/m/#!topic/tm/85Kqi5bKtHo
[10] N3862: Towards a Transaction-safe C++ Standard Library: std::list, 2014-01-20, LEWG, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3862.pdf