Document Number: N3862
Date: 2014-01-20
Authors: Justin Gottschlich, justin.gottschlich@intel.com
Michael Spear, spear@cse.lehigh.edu
Michael Wong, michaelw@ca.ibm.com
with other members of the transactional memory study group (SG5), including:
Hans Boehm, hans.boehm@hp.com
Victor Luchangco, victor.luchangco@oracle.com
Jens Maurer, jens.maurer@gmx.net
Maged Michael, magedm@us.ibm.com
Mark Moir, mark.moir@oracle.com
Torvald Riegel, triegel@redhat.com
Michael Scott, scott@cs.rochester.edu
Tatiana Shpeisman, tatiana.shpeisman@intel.com
Project: Programming Language C++, LEWG, SG5 Transactional Memory
Reply to: Michael Wong michaelw@ca.ibm.com (Chair of SG5)
Revision: 1

Towards 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.

1. Introduction

This paper describes our first effort to transactionalize, that is, to enable transactional memory (TM) support in the C++ Standard Library starting with a single container, based on the syntax as described in 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 (N3862) shows 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.

2. 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.

In addition, 23.3.5.6 indicates that `std::swap` is specialized for `std::list`. Thus while no code modifications were required to support its use, the specification will require any such specializations of std::swap for std::list to be `transaction_safe`.

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

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);
  };
}
```

2. Update 23.3.5.6/1 [list.special] to add transaction_safe keyword to swap

23.3.5.6 list specialized algorithms [list.special]

```
template <class T, class Allocator>
  void swap(list<T,Allocator>& x, list<T,Allocator>& y) transaction_safe;
```

1 *Effects:*

```
    x.swap(y);
```

4. Discussion

In the process of transactionalizing `std::list`, we considered three issues.

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 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 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.

5. 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
```

6. Acknowledgements

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

7. Reference

[1]N3859, Transactional Memory Support for C++, 2014-01-20, SG5, new paper
[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