Towards Transactional Memory for OpenMP

Michael Wong*, Eduard Ayguade**,
Justin Gottschlich***, Victor Luchangco†, Bronis R. de Supinski‡, Barna Bihari‡, and
other members of the WG21 SG5 Transactional Memory Sub-Group

Abstract. The OpenMP specification lacks a shared memory concurrency mechanism that is composable. None of the OpenMP concurrency mechanisms, such as OMP critical, locks, or atomics support composition. In this paper, we motivate the need for transactional memory (TM) in OpenMP chiefly to support composition of realistic programs. However, we also consider whether TM is easier to program than locks; the use-case for TM; and whether a software-only TM can outperform traditional locking through a survey of recent publications. This paper advances upon previous proposals of OpenMP TM by introducing a new construct specifically to handle irrevocable actions, which is also composable. It also proposes a pure atomic transaction construct as well as the concept of transaction safety. Further, we examine how our proposed construct integrates with current OpenMP constructs.

1 Introduction

Locks and atomics have serious weaknesses when constructing larger programs out of smaller pieces; they are often described as not being composable. Because locks and atomics are so difficult to compose, they do not support modular programming well [1]. As multithreaded programs increase in size and complexity, more advanced abstractions will be needed to mitigate the programming complexity that naturally arises from frequent use of synchronization in large-scale software systems. We present transactional memory (TM) for OpenMP and show that TM provides stricter correctness guarantees than other OpenMP concurrency techniques and may be easier to use.

2 Limitations of Current Concurrency Techniques

OpenMP V4.0, the latest release [2], currently includes four synchronization mechanisms: locks, barriers, atomics and critical sections [3]. These mechanisms synchronize objects in shared memory but are unnecessarily limit performance or can be challenging to use properly. TM provides greater flexibility and ease of use and in the case of template programming, or callback-style programming, TM offers correctness that none of the other constructs can offer.

Mutual exclusion as implemented as critical sections is perhaps the most common form of concurrency control for shared memory parallel programming. In general, mutual exclusion ensures program correctness by limiting access to shared memory variables to one thread at a time. Mutual exclusion achieves this restriction by using mutually exclusive locks, also known as OpenMP locks or critical sections. For a thread to

* IBM, **BSC, ***Intel, †Oracle, ‡LLNL

access a shared memory variable, it must first acquire the lock that protects the shared memory variable. When a thread has completed its access to the shared memory variable, it releases the lock. On the surface, the concept of mutual exclusion is straightforward and easy to apply. However, the majority of the parallel programming community agrees that mutual exclusion quickly becomes unwieldy if used in large-scale software. Furthermore, many experts believe it is notoriously challenging to write both correct and efficient large-scale multithreaded software using mutual exclusion [4].

A less frequent form of synchronizing access to shared memory variables is to use non-blocking atomic primitives which is how most OpenMP barriers or atomics are implemented, such as compare-and-swap (CAS) or load-linked store-conditional (LL/SC), or more recently, C++ and OpenMP atomic types to write non-blocking algorithms. Although these approaches can often yield substantial concurrency, their non-trivial use is often limited to only expert parallel programmers as correctly building even simple data structures, such as a queue, using these kinds of synchronization primitives can be challenging [4].

Another possible speculative technique, known as lock elision, has been proposed where locks are elided, that is, not acquired, in cases where conflicts in the data they protect are absent. Lock elision may reduce unnecessary synchronization, thereby increasing concurrent throughput. However, while lock elision can result in notable performance gains over lock-based systems that do not use lock elision, it still requires that programmers write correct and complete locking schemes for their multithreaded software. Therefore, lock elision does not significantly reduce, nor is it intended to reduce, the challenge of writing multithreaded software.

3 Motivation for Transactional Memory in OpenMP

Synchronizations supported by OpenMP are OpenMP critical sections, barrier, mutexes or locks, and atomics as described in our previous papers [5], [6]. Locks and atomics are basic abstractions used to access and mutate shared state, as defined by the Three Pillars of Concurrency [1]. From this view, these abstractions aim to avoid data races and synchronize objects in shared memory. Unfortunately, locks and atomics are notoriously difficult to use [7]. Simple coarse-grain locking strategies, where all program data are protected using one or few locks, lead to unnecessary serialization of program execution and loss of performance. Sophisticated fine-grain locking or use of atomics result in complex association between data and synchronization that protects access to that data. A failure to correctly maintain these associations throughout the program leads to concurrency errors, such as data races and deadlocks. Moreover, synchronization strategies designed to work well on one platform often perform poorly on a platform with a different number of hardware threads or a different cost for synchronization primitives.

When a thread calls two functions in a sequence, without protecting such calls with some form of synchronization, other threads may be able to interleave instructions between the two calls. When this occurs, other threads can perceive each call as a distinct action, even if each function appears to execute atomically on its own. To prevent this behavior, one usually wraps the entire sequence with a lock. As a result, when we com-

bine function calls into a larger program, we often end up with nested lock acquisitions: a function acquires a lock while one of its callers holds another one. Unfortunately, when multiple threads acquire more than one lock at a time, the program may deadlock if the threads do not follow a consistent locking order across all threads. The usual advice is to acquire locks only in a predetermined order, but there appears to be no practical way to enforce such an ordering in a large software system. Edward Lee [8] presents a simple practical example of this kind of problem (the observer pattern) in a Java context, but it applies equally well to C++. Implementing this simple pattern with locks turns out to be nearly impossible. In spite of Lees technical report title (i.e., The Problem With Threads), the problem he describes actually lies with locks rather than threads, as a concurrent transaction-based solution is not significantly harder than a sequential one.

Functions that use atomics to perform shared memory operations are also not composable, but in a different sense: atomics provide no easy mechanism for combining the small atomic actions into larger atomic actions. The atomic operations are not intended to compose into bigger atomic operations. Parallel programming inherently entails increased complexity in developing programs, in reasoning about programs, and in reproducing bugs. By using a transaction—rather than one or more locks—to synchronize a section of code, the programmer is not required to specify which metadata (i.e., which named variable) is used to synchronize data. In addition to simplifying the resulting software, this approach alleviates the need of a fixed order of execution, which is required by locks to avoid deadlock. This results in simpler designs that are easier to write, reason about, and maintain. Furthermore, it enables specialized synchronization support for different platforms, which can be improved over time, without requiring changes to the application code.

To address these concerns, we propose that OpenMP be extended to include transactional language constructs, or for short, transactional memory (TM). Our proposal and its integration into OpenMP are described in detail in Section 4 below. Some of the key benefits of TM, compared to locks, can be found in our paper that convinced the C++ Standadard committee [9] to start a New Feature Proposal.

Extending OpenMP to include TM will improve the modularity of concurrent libraries, make OpenMP easier to teach and learn, and supply a programming model for architectures based on IBM's current Blue Gene/Q and Power8, and Intel's Haswell RTM. This is especially important as OpenMP moves further into commercial applications as well as remaining relevant in scientific workloads.

3.1 Survey of Prior Work supporting TM use cases, usability and performance in real world applications

A number of recent publications report experiences gathered in the effort of parallelizing realistic applications using transactional memory, usually driving conclusions about the programmability/performance trade-off. Usually they are based on the use of software implementations of transactional memory, with varying results in terms of performance. Examples of such efforts include Delaunay triangulation [10], minimum spanning forest of sparse graphs [11] Lee routing algorithm [12], multiplayer game servers such as QuakeTM [13] and Atomic Quake [14] (based on a lock-based version

of Quake [15]) and SynQuake [16]; or benchmarks (STMBench7 [17], STAMP [18] and RMS-TM [19], all of them composed of a number of applications representative of a variety of application domains.

Several of these studies show an initial answer to the question of performance. Is Software Transactional Memory (STM) faster then locks in a real world application, and not just toy laboratory benchmarks? SynQuake [16] used a form of Quake, a game server reimplemented using pure Software TM (STM) from locks to examine the performance and scalability difference of TM without hardware support. Parallelization of multi-player game code for the purposes of scaling the game server is inherently difficult. Game code is typically complex, and can include use of spatial data structures for collision detection, as well as other dynamic artifacts that require conservative synchronization. The nature of the code may thus induce substantial contention due to false sharing, as well as true sharing between threads, in a parallel lock-based game implementation [15]. Each Quake player action usually includes dynamically evolving sub-actions; a person may move while shifting items in their backpack, throwing an object at a distance, grabbing a nearby object, and/or shooting, which together constitute a single player action. Since the terrain within the potentially affected area may contain mutable objects, all sub-actions need to be processed together as an atomic, consistent unit for the purposes of collision detection with other player actions.

Thus, conservatively acquiring all locks at the beginning of the the action induces unnecessary conflicts, by locking more objects than necessary, and unnecessarily long conflict duration by holding these locks for longer periods than needed. Fine-grain locking of the action sequence is not even possible as it leads to no atomicity of the action, and leads to problems with deadlocks and inconsistent views. In contrast, using Transactional Memory support by implementing player actions (i.e split actions into subactions), and track accesses to shared and private data using conflict detection and resolution, the atomicity and consistency of the whole player action is automatically provided by the underlying transactional support. They showed that STM support results in reduced false sharing overall, in terms of both number of conflicts and duration of conflicts. The transaction simply commits if there is no conflict with another player, or rolls back if conflict occurs. The result was that STM was about 33 percent faster than locks for a 4-8 thread medium contention case and STM scales better in all cases of low, medium and high contention.

On usability, other studies have been conducted trying to quantify the claim that concurrent programming with transactional memory is easier than using other alternatives such as locks by the Rossbach and Pankratius studies [20], [21]. In Rossbach [20] where they asked students to program in three different ways: coarse grained-locks, fine-grained locks and TM. There was firm evidence that fine-grained locking tasks were more likely to contain errors than coarse-grained or TM. The most common errors were acquiring a lock and never releasing them. Students found that TM was still harder to program (because of a lack of TM documentation at that time) than coarse-grained locks but easier than fine-grained locks. In Pankratius [21], they created separate teams working on locks and on TM to work on a search application. The average Lines of Code (LOC) were the same across all teams, but the TM teams had fewer LOC with parallel constructs than the lock teams. They found that TM allowed teams

to think more sequentially and spent less time as the locks teams on writing parallel code before moving on to performance testing. Yet one of the TM team had the first working parallel version, even though they subjectively believed they advanced slowly. By project deadline, all the lock teams while completing the functionality, had performance problems with one lock team not finishing. One of the TM teams was deemd to have the best performance, while another TM team's code did not work at all despite a member on that team having commercial programming experience. TM still requires good programmer and is not a panacea to parallel programming difficulties. But it does hold promise as being easier to use than fine-grained locks in very large and complex parallel programming tasks.

Two proposals were presented almost at the same time [22], [6] with the aim of adding transactional memory support within the OpenMP programming interface. The topic has reborn with a more recent proposal [3], presenting results using hardware transactional memory, which can significantly reduce the complexity of shared memory programming while retaining efficiency. This work extended a previous work [5] that already demonstrated that even with the relatively high overheads of software implementations, transactions could outperform OpenMP critical sections.

While examining use cases, Gottschlich and Boehm [23] asserted that TM is actually necessary for functional correctness for generic programming in C++ because the layering and composition of library software means that there is no way for the client of a template which also use locks to know what locks are being held by the caller. They debunked the popular belief that enforced locking ordering can avoid deadlock and show that this is essentially impossible with C++ template programming. This is a form of callback-style programming that exists in C and Fortran and takes the question of whether TM is useful beyond merely performance and scalability, but firmly moves it to that it is needed for correctness of today's programming paradigms.

The paper then takes this one step further by showing what makes generic programming different from prior examples is that many or most of the function calls and operator invocations depend on type parameters, and are thus effectively callbacks. That might include the C++ assignments operator, and the constructor. It also includes the syntactically invisible destructor, and even possibly the syntactically invisible construction and destruction of expression temporaries. These constructors and destructors are likely to acquire locks if, for example, the constructor takes possession of resources from a shared free list that are returned by the destructor. In order to enforce a lock ordering, the author of any generic function acquiring locks (or that could possibly be called while holding a lock) would have to reason about the locks that could potentially be acquired by any of these operators, which appears thoroughly intractable.

Their conclusion is either to forbid locks in templates and callbacks, or allow TM to re-enable generic or callback style programming.

In 2008, IBM, Intel and Sun (later acquired by Oracle) started joint teleconference discussions every other week to design such a common language [24]. HP, Redhat and other members from academia later joined the initial group. In 2009, version 1.0 of this language was released and was followed by version 1.1 in 2011 which added support for exceptions for C++. The reason C++ was chosen as the language to bolt on TM was because it has the most complex language features, in terms of polymorphism,

exceptions, and memory model. In 2012, this proposal was brought to C++ Standard which after examining many of the use cases, usability and performance claims, felt it was justified to prepare for adoption of this high-level language, by starting work immediately. C++ Standard formed Study Group 5 (SG5) [25] lead by one of the co-authors to develop a proposal. In 2014, after two more years of collaboration to better fit TM into the C++ language with contribution from the creator of C++, it is nearing completion with Standard wording not far behind.

This proposal takes from the SG5 proposal [26] which is only for C++, merges with the BSC OpenMP proposal [6], while using the experience from IBM's BG/Q HTM design [27] and adapts it to existing OpenMP Language to offer an initial design for TM in OpenMP for the future that works on C, C++ , and Fortran. It is appropriate as the basis for an OpenMP Working Group (WG), since all the SG5 members are also OpenMP members. This WG would further develop this proposal with the goal of an OpenMP Technical Report to gain more implementation experience and user feedback.

4 A proposal for an OpenMP Transactional Memory Technical Report

We introduce two kinds of blocks to exploit transactional memory: *synchronized blocks* in Section 4.2 and *atomic blocks* called *OMP transaction* (as a keyword placeholder) in Section 4.1. Synchronized blocks behave as if all synchronized blocks were protected by a single global recursive mutex. Atomic blocks (also called *atomic transactions*, or just *transactions*) appear to execute atomically and not concurrently with any synchronized block (unless the atomic block is executed within the synchronized block).

Some operations are prohibited within atomic blocks because it may be impossible, difficult, or expensive to support executing them in atomic blocks; such operations are called *transaction-unsafe*. An atomic block also specifies how to handle an exception thrown but not caught within the atomic block.

Some noteworthy points about synchronized and atomic blocks:

Data races Operations executed within synchronized or atomic blocks do not form data races with each other. However, they may form data races with operations not executed within any synchronized or atomic block. As usual, programs with data races have undefined semantics.

Exceptions When an exception is thrown but not caught within an atomic block, the effects of operations executed within the block may take effect or be discarded, or terminate may be called. This behavior is specified by an additional keyword in the atomic block statement, as described in Section 4.1. An atomic block whose effects are discarded is said to be *canceled*. An atomic block that completes without its effects being discarded, and without calling terminate, is said to be *committed*.

Transaction-safety As mentioned above, transaction-unsafe operations are prohibited within an atomic block. As a practical matter, some code is considered transaction-unsafe because we do not know effective ways to execute it atomically without special hardware support. This restriction applies not only to code in the body of an atomic block, but also to code in the body of functions called (directly or indirectly) within the atomic block. To support static checking of this restriction, we

introduce pragmas to declare that a function or function pointer is transaction-safe, and augment the type of a function or function pointer to specify whether it is transaction-safe. We also introduce a pragma to explicitly declare that a function is *not* transaction-safe.

To reduce the burden of declaring functions transaction-safe, a function is assumed to be transaction-safe if its definition does not contain any transaction-unsafe code and it is not explicitly declared transaction-unsafe. Furthermore, unless declared otherwise, a non-virtual function whose definition is unavailable is assumed to be transaction-safe. (This assumption does *not* apply to virtual functions because the callee is not generally known statically to the caller.) These assumptions are checked at link time.

4.1 Atomic Blocks

This is a pure form of a transaction and is based on combining the C++ SG5 [28] proposal and BSC's OpenMP TM extension proposal [6].

An *atomic block* can be written in one of the following forms:

#pragma omp *transaction [clause[[,] clause]...]* { *body* }

The clause following *transaction* can specify the atomic block's *exception specifier*. It specifies the behavior when an exception escapes the transaction or an OpenMP cancel atomic occurs within the TM region:

- *noexcept:* This is undefined behavior and is not allowed; no side effects of the transaction can be observed.
- *commitonesc:* The transaction is committed and the exception is thrown.
- *cancelonesc:* If the exception is transaction-safe (defined below), the transaction is canceled and the exception is thrown. Otherwise, it is undefined behavior. In either case, no side effects of the transaction can be observed.

Code within the body of a transaction must be *transaction-safe* (i.e. it must not be transaction-unsafe). Code is *transaction-unsafe* if:

- it contains an initialization of, assignment to, or a read from a volatile object;
- it is a transaction-unsafe `asm` declaration (the definition of a transaction-unsafe `asm` declaration is implementation-defined); or
- it contains a call to a transaction-unsafe function, or through a function pointer that is not transaction-safe

While we have pragma syntax to allow declaring and defining functions for transaction safety, we will not show it here due to space constraints.

Synchronization via locks and atomic objects is not allowed within atomic blocks (operations on these objects are calls to transaction-unsafe functions in the current proposal, but may be relaxed in future revision of the TR).

Jumping into the body of an atomic block using `goto` or `switch` is prohibited.

The body of an atomic block appears to take effect atomically: no other thread sees any intermediate state of an atomic block, nor does the thread executing an atomic block see the effects of any operation of other threads interleaved between the steps within the atomic block.

The evaluation of any atomic block synchronizes with every evaluation of any atomic or synchronized block by another thread, so that the evaluations of non-nested atomic and synchronized blocks across all threads are totally ordered by the synchronizes-with relation. Thus, a memory access within an atomic block does not race with any other memory access in an atomic or synchronized block. However, a memory access within an atomic block may race with conflicting memory accesses not within any atomic or synchronized block. The exact rules for defining data races are defined by the memory model [26].

As usual, programs with data races have undefined semantics.

Although it has no observable effects, a canceled atomic block may still participate in data races.

This proposal provides "closed nesting" semantics for nested atomic blocks.

Use of atomic blocks Atomic blocks are intended in part to replace many uses of mutexes for synchronizing memory access, simplifying the code and avoiding many problems introduced by mutexes (e.g., deadlock). We expect that some implementations of atomic blocks will exploit hardware and software transactional memory mechanisms to improve performance relative to mutex-based synchronization. Nonetheless, programmers should still endeavor to reduce the size of atomic blocks and the conflicts among atomic blocks and with synchronized blocks: poor performance is likely if atomic blocks are too large or concurrent conflicting executions of atomic and synchronized blocks are common. **Example**

The following code illustrates with a bank account example the atomicity of atomic. blocks.

```
1 class Account {
2    int bal;
3  public:
4    Account(int initbal) { bal = initbal; };
5
6    void deposit(int x) {
7      #pragma omp transaction noexcept {
8        this.bal += x;
9      }
10   };
11
12   void withdraw(int x) {
13     deposit(-x);
14   };
15
16   int balance() { return bal; }
17 }
18
19 void transfer(Account a1, a2; int x;) {
20   #pragma omp transaction noexcept {
21     a1.withdraw(x);
22     a2.deposit(x);
23   }
24 };
25
26 Account a1(0), a2(100);
```

```
Thread 1                    Thread 2
--------                    --------

transfer(a1, a2, 50);       #pragma omp transaction noexcept {
                              r1 = a1.balance() + a2.balance();
                            }
                            assert(r1 == 100);
```

The assert cannot fire, because the transfer happens atomically and the two calls to `balance` happen atomically.

Example demonstrating need for transaction_cancelonesc Here, we extend the above example slightly so that transactions are logged by a function that may throw an exception, for example due to allocation failure.

```
void deposit(int x) {
  #pragma omp transaction cancelonesc {
    log_deposit(x);         // might throw
    this.bal += x;
  }
}

void withdraw(int x) {
  deposit(-x);
}

void transfer(account a1, a2; int x;) {
  try {
    #pragma omp transaction cancelonesc {
      a1.withdraw(x);
      a2.deposit(x);
    } catch (...) {
      printf("Transfer failed");
    }
  }
}
```

If the call from `transfer()` to `a2.deposit()` throws an exception, we should not simply commit the transaction, because the withdrawal has happened but the deposit has not. Canceling the transaction provides an easy way to recover to a good state, without violating the invariant the transaction in `transfer()` is intended to preserve. In this simple example, an error message is printed indicating that the transfer did not happen.

Default behavior The default for atomic transactions without any of the three clauses (noexcept, commitonesc, cancelonesc) is as if the user wrote *cancelonesc*. This offers a pure transaction that rolls back. The other two optional clauses (noexcept and commitonesc) do not rollback and therefore offer no invariance protection. But they do still offer advanced synchronization ability. However, they are still limited in that they cannot have any transaction unsafe actions. We show in the next section how to handle transactions with transaction unsafe actions.

4.2 Synchronized Blocks

The synchronized blocks variant is a simple replacement for locks that is composable and offers only a synchronization ability with no invariance protection. Furthermore,

synchronized blocks can become irrevocable in the presence of unsafe actions and that distinguishes it from an atomic transaction.

A *synchronized block* has the following form:

#pragam omp `synchronized` { *body* }

The evaluation of any synchronized block synchronizes with every evaluation of any synchronized block (whether it is an evaluation of the same block or a different one) by another thread, so that the evaluations of non-nested synchronized blocks across all threads are totally ordered by the synchronizes-with relation as defined by C++ and Java memory model. That is, the semantics of a synchronized block is equivalent to having a single global recursive mutex that is acquired before executing the body and released after the body is executed (unless the synchronized block is nested within another synchronized block). Thus, an operation within a synchronized block never forms a data race with any other operation within a synchronized block (the same block or a different one).

Entering and exiting a nested synchronized block (i.e., a synchronized block within another synchronized block) has no effect.

Jumping into the body of a synchronized block using `goto` or `switch` is prohibited.

Use of synchronized blocks Synchronized blocks are intended in part to address some of the difficulties with using mutexes for synchronizing memory access by raising the level of abstraction and providing greater implementation flexibility [23] With synchronized blocks, a programmer need not associate locks with memory locations, nor obey a locking discipline to avoid deadlock: Deadlock cannot occur if synchronized blocks are the only synchronization mechanism used in a program.

Although synchronized blocks can be implemented using a single global mutex, we expect that some implementations of synchronized blocks will exploit recent hardware and software mechanisms for transactional memory to improve performance relative to mutex-based synchronization. For example, threads may use speculation and conflict detection to evaluate synchronized blocks concurrently, discarding speculative outcomes if conflict is detected. Programmers should still endeavor to reduce the size of synchronized blocks and the conflicts between synchronized blocks: poor performance is likely if synchronized blocks are too large or concurrent conflicting evaluations of synchronized blocks are common. In addition, certain operations, such as I/O, cannot be executed speculatively, so their use within synchronized blocks may hurt performance.

Example

The following example illustrates synchronized blocks and non-races between accesses within transactions (including synchronized blocks). Suppose we add the following method to the `Account` class shown in Section 4.1.

```
1 void print_balances_and_total (account a1, a2) {
2   #pragma omp synchronized {
3     printf("First account balance: %ld", a1.balance());
4     printf("Second account balance: %ld", a2.balance());
5     printf("Total: %ld", a1.balance() + a2.balance());
6   }
7 }
```

Observations:

- This program is data-race-free: all concurrent accesses are within transactions.
- The synchronized block cannot be replaced with an atomic block, as I/O is not transaction-safe (due to calls to `printf`, which is a transaction-unsafe function).
- Balances will be consistent and total will equal sum of balances displayed.
- If we eliminate the synchronized block from this example (so the calls to `balance()` in `print_balances_and_total()` are not in transactions), then this program is racy.

4.3 Nesting of OpenMP parallel regions and Transaction Blocks

In the common case of a TM region nested inside an OpenMP parallel region, the outer OpenMP region is run in parallel and the TM region is run speculatively. In the opposite case where an OpenMP parallel region is nested inside a TM region, there are several choices which needs to be debated within the community.

Currently on IBM's Blue Gene/Q system [27], an OpenMP region running in parallel inside the speculative TM region causes the TM region to be stopped. The stopped transaction is then rolled back and run nonspeculatively. The inner OpenMP region is run nonspeculatively by multiple threads. This is considered to be quite restrictive and heavy weight. An alternative is where the transaction could be executed *as if* the OpenMP portion was serialized. This could have complication with hardware and if the user create a race condition inside the transaction, it would be caveat emptor.

Another choice is that the parallel region inside the TM region can be executed with one thread. This solution will often be better than restarting the transaction and running it non-speculatively. There will be complication if the OpenMP region do some undesirable action such as checking for the number of threads being more than one. But these are details that can be worked out in committee.

4.4 Interaction between OpenMP worksharing/tasking constructs and Transaction Blocks

We also intend to introduce interaction of TM with existing OpenMP constructs. These are now called composite constructs as they enable additional semantics. Starting with the workshare constructs, we propose the following where each iteration of the loop constitutes an atomic transaction with the usual clauses available.

```
1 #pragma omp for transaction
2 for (   ;   ;   )
3 { ... }
```

Similar for an OpenMP section construct where each section is an atomic transaction.

```
1 #pragma omp sections transaction
2 #pragma omp section
3 { ... }
4 #pragma omp section
5 { ... }
```

We also plan to support TM with OpenMP tasks. Tasks are defined as deferrable units of work that can be executed by any thread in the thread team associated to the active parallel region. Task can create new tasks and can also be nested inside work-sharing constructs. In this scenario, data access ordering and synchronization based on locks will be even more difficult to express, so transactions appear as an easy way to express intent and leave the mechanisms to the TM implementation. For tasks we propose tagging a task as a transaction, using the same clause specified above.

```
1 #pragma omp task transaction
2 { ... }
```

We will also need consideration of the interaction with cancellation constructs. These are details to be explored in future proposals and in committee.

4.5 Memory Model and Race Free Semantics

Transactions impose ordering constraints on the execution of the program. In this regard, they act as synchronization operations similar to the synchronization mechanisms defined in the C++11 standard (i.e., locks and C++11 atomic variables). The C++11 standard defines the rules that determine what values can be seen by the reads in a multi-threaded program. Transactions affect these rules by introducing additional ordering constraints between operations of different threads.

An execution of a program consists of the execution of all of its threads. The operations of each thread are ordered by the sequenced before relationship that is consistent with each threads single threaded semantics. The C++11 library defines a number of operations that are specifically identified as synchronization operations. Synchronization operations include operations on locks and certain atomic operations (that is, operations on C++11 atomic variables). In addition, there are memory_order_relaxed atomic operations that are not synchronization operations. Certain synchronization operations synchronize with other synchronization operations performed by another thread. (For example, a lock release synchronizes with the next lock acquire on the same lock.)

The sequenced before and synchronizes with relationships contribute to the happens before

1. If an operation A is sequenced before an operation B then A happens before B.
2. If an operation A synchronizes with an operation B then A happens before B.
3. If there exists an operation B such that an operation A happens before B and B happens before

Two operations conflict if one of them modifies a memory location and the other one accesses or modifies the same memory location. The execution of a program contains a data race if it contains two conflicting operations in different threads, at least one of which is not an atomic operation, and neither happens before the other. Any such data race results in undefined behavior. A program is race-free if none of its executions contain a data race. In a race-free program each read from a non-atomic memory location sees the value written by the last write ordered before it by the happens-before relationship. It follows that a race-free program that uses no atomic operations with memory

ordering other than the default memory_order_seq_cst behaves according to one of its sequentially consistent executions.

Outermost transactions (that is, transactions that are not dynamically nested within other transactions) appear to execute sequentially in some total global order that contributes to the synchronizes with relationship. Conceptually, every outermost transaction is associated with StartTransaction and EndTransaction operations, which mark the beginning and end of the transaction. A StartTransaction operation is sequenced before all other operations of its transaction. All operations of a transaction are sequenced before its EndTransaction operation. Given a transaction T, any operation that is not part of T and is sequenced before some operation of T is sequenced before Ts StartTransaction operation. Given a transaction T, Ts EndTransaction operation is sequenced before any operation A that is not part of T and has an

There exists a total order over all StartTransaction and EndTransaction operations called the executed by different threads do not interleave. In other words, transactional synchronization order is such that a StartTransaction operation executed by one thread does not occur in between a matching pair of StartTransaction and EndTransaction operations executed by another thread.

The transactional synchronization order contributes to the synchronizes with relationship defined in the C++11 standard. In particular, each EndTransaction operation synchronizes with the next StartTransaction operation in the transactional synchronization order executed by a different thread.

The definition of the synchronizes with relation affects all other parts of the memory model, including the definition of the happens before relationship, visibility rules that specify what values can be seen by the reads, and the definition of data race freedom. Consequently, including transactions in the synchronizes with relation is the only change to the memory model that is necessary to account for transaction statements. With this extension, the C++11 memory model fully describes the behavior of programs with transaction statements.

The C++11 memory model has consequences for compiler optimizations.Sequentially valid source-to-source compiler transformations that transform only code between synchronization operations (which include StartTransaction and EndTransaction operations), and which do not introduce data races, remain valid. Source-to-source compiler transformations that introduce data races (e.g., hoisting load operations outside of a transaction) may be invalid depending on a particular implementation of this specification.

5 Future OpenMP Recommendation

We propose an OpenMP Transactional Memory Technical Report (TR), to enable early implementation experience and obtain feedbacks from the community. Transactional Memory forms a key cornerstone of tools for synchronization that enables composability whereas critical sections, mutexes, locks, atomics, even lock elision cannot. It enables functional correctness in C and Fortran call back programming style and C++ generic programming. Recent surveys have some data point showing it is easier to use than fine-grained locks, and some real-world tests have shown even an STM imple-

mentation can scale and perform better than fine-grained locks. As such, it enables and simplifies support for large scale programs that contain complex locking semantics.

As part of the OpenMP Technical Report process which was introduced in 2012 to give OpenMP more agility to publish early directions, it is non-normative (i.e. not part of the ratified OpenMP specification). If it is deemed that it is useful, and sufficient user feedback supports its ratification, then that will be determined in future.

Furthermore, this proposal is agnostic to hardware. OpenMP cannot legislate requirement for TM hardware. This proposal can be entirely implemented in software, hardware, some hybrid or adaptive form of TM.

Our next goal is to provide an implementation using BSC's Mercurium OpenMP compiler [6] or GNU compiler to demonstrate the concept and confirm the performance capability and suitability for generic programming and callback-style programming.

References

1. Sutter, H.: The Pillars of Concurrency. Dr. Dobbs (July 2007)
2. OpenMP ARB: OpenMP Application Program Interface, v. 4.0 (June 2013)
3. Bihari, B.L., Wong, M., Wang, A., de Supinski, B.R., Chen, W.: A case for including transactions in openmp ii: Hardware transactional memory. In: Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World IWOMP'12. (2012) 44–58
4. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (March 2008)
5. Wong, M., Bihari, B.L., de Supinski, B.R., Wu, P., amd Y. Liu, M.M., Chen, W.: A case for including transactions in OpenMP. In: IWOMP 2010 Conference Proceedings, Tsukuba, Japan, LNCS 6132 (June 2010) 149–160
6. Milovanovic, M., Ferrer, R., Unsal, O., Cristal, A., Ayguade, E., Labarta, J., Valero, M.: Transactional memory and openmp. In: Proceedings International Workshop on OpenMP IWOMP-2007. (2007) 37–53
7. Sutter, H.: The Trouble with Locks. Dr. Dobbs (March 2005)
8. MPI Forum: The Problem with Threads. Technical report, Electrical Engineering and Computer Sciences University of California at Berkeley (January 2006)
9. Wong, M., Boehm, H., Gottschlich, J., Shpeisman, T.: Transactional Language Constructs for C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3341.pdf (Jan 2012)
10. Scott, M.L., Spear, M.F., Daless, L., Marathe, V.J.: Delaunay triangulation with transactions and barriers. In: Proceedings IEEE International Symposium on Workload Characterization. (2007)
11. Kang, S., Bader, D.A.: An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPoPP '09, Civil-Comp Press (2009) 15–24
12. Ansari, M., Kotselidis, C., Jarvis, K., Lujan, M., Kirkham, C., Watson, I.: Lee-tm: A non-trivial benchmark for transactional memory. In: Proceedings of the 7th International Conference on Algorithms and Architectures for Parallel Processing ICA3PP '08. (2008)
13. Gajinov, V., Zyulkyarov, F., Unsal, O.S., Cristal, A., Ayguade, E., Harris, T., Valero, M.: Quaketm: Parallelizing a complex sequential application using transactional memory. In: Proceedings of the 23rd International Conference on Supercomputing ICS '09. (2009) 126–135

14. Zyulkyarov, F., Gajinov, V., Unsal, O.S., Cristal, A., Ayguade, E., Harris, T., Valero, M.: Atomic quake: Using transactional memory in an interactive multiplayer game server. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. (2009) 25–34
15. Abdelkhalek, A., Bilas, A.: Parallelization and performance of interactive multiplayer game servers. In: Proceedings of the 18th International Parallel and Distributed Processing Symposium, IPDPS. (2004)
16. Lupei, D., Simion, B., Bogdan, Pinto, D., Misler, M., Burcea, M., Krick, W., C. Amza, C.: Transactional memory support for scalable and transparent parallelization of multiplayer games. In: Proceedings of the 5th European Conference on Computer Systems EuroSys '10. (2010) 41–54
17. Guerraoui, R., Kapalka, M., Vitek, J.: Stmbench7: A benchmark for software transactional memory. In: Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems EuroSys '07. (2007) 315–324
18. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: Proceedings of The IEEE International Symposium on Workload Characterization IISWC '08. (2008) 315–324
19. Kestor, G., Stipic, S., Unsal, O., Cristal, A., Valero, M.: Rms-tm: A transactional memory benchmark for recognition, mining and synthesis applications. In: Proceedings 4th ACM SIGPLAN Workshop on Transactional Computing TRANSACT. (2009)
20. Rossbach, C.J., Hofmann, O.S., Witchel, W.: Is transactional programming actually easier? In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPoPP '10. (2010) 47–56
21. Pankratius, V., Adl-Tabatabai, A.: A study of transactional memory vs. locks in practice. In: Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures SPAA '11. (2011) 43–52
22. Baek, W., Minh, C.C., Trautmann, M., Kozyrakis, C., Olukotun, K.: The opentm transactional application programming interface. In: Proceedings International Conference on Parallel Architectures and Compilation Techniques PACT-2007. (2007) 376–387
23. Gottschlich, J.E., Boehm, H.J.: Generic programming needs transactional memory. In: The 8th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT). (2013)
24. Group, T.M.S.D.: https://sites.google.com/site/tmforcplusplus/. https://sites.google.com/site/tmforcplusplus/ (May 2014)
25. Wong, M., Gottschlich, J.: SG5: Software Transactional Memory (TM) Status Report. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3422.pdf (Sept 2012)
26. Luchangco, V., Wong, M.: Transactional Memory Support for C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3919.pdf (Feb 2014)
27. IBM: IBM XL C/C++ for Transactional Memory for AIX, V0.9 Language Extensions and Users Guide. http://dl.alphaworks.ibm.com/technologies/xlcstm/xlcstm-whitepaper.pdf (May 2008)
28. Sutter, H.: https://isocpp.org/std/status. https://isocpp.org/std/status (May 2014)