Transactional Memory and OpenMP

Miloš Milovanović, Roger Ferrer, Osman S. Unsal, Adrian Cristal, Xavier Martorell, Eduard Ayguadé, Jesús Labarta, and Mateo Valero

Barcelona Supercomputing Center
c/ Jordi Girona 31, 08034 Barcelona, Spain
{milos.milovanovic,roger.ferrer,osman.unsal,adrian.cristal, xavier.martorell,eduard.ayguade,jesus.labarta,mateo.valero}@bsc.es
http://www.bsc.es

Abstract. Future generations of Chip Multiprocessors (CMP) will provide dozens or even hundreds of cores inside the chip. Writing applications that benefit from the massive computational power offered by these chips is not going to be an easy task for mainstream programmers who are used to sequential algorithms rather than parallel ones. This paper explores the possibility of using Transactional Memory (TM) in OpenMP, the industrial standard for writing parallel programs on shared-memory architectures, for C, C++, and Fortran. One of the major complexities in writing OpenMP applications is the use of critical regions (locks), atomic regions and barriers to synchronize the execution of parallel activities in threads. TM has been proposed as a mechanism that abstracts some of the complexities associated with concurrent access to shared data while enabling scalable performance. The paper presents a first proof-of-concept implementation of OpenMP with TM. Some extensions to the language are proposed to express transactions. These extensions are handled by our source-to-source OpenMP Mercurium compiler and our Software Transactional Memory (STM) library Nebelung that supports the code generated by Mercurium. The current implementation of the library has no support at the hardware level, so it is a proof-of-concept implementation. Hardware Transactional Memory (HTM) or Hardware-assisted STM (HaSTM) are seen as possible paths to make the tandem TM-OpenMP more usable. The paper finishes with a set of open issues that still need to be addressed, either in OpenMP or in the hardware/software implementations of TM.

Keywords: Compiler, OpenMP, Software Transaction Memory, STM Library.

1 Introduction

The trend towards incorporating more cores in Chip Multiprocessors (CMP) will continue, with the potential for hundreds of cores for future technology generations. Inefficient data access ordering and synchronization primitives such as locking will limit programmer productivity and application performance for those future processors. Transactional Memory (TM) is a crucial mechanism to tackle this problem by abstracting away the complexities associated by concurrent access to shared data [1] where multiple threads need to simultaneously access shared memory locations atomically.

OpenMP [2], the industrial standard for writing parallel programs on shared-memory architectures, for C, C++, and Fortran is a "traditional" programming model in terms of mechanisms offered to guarantee mutual exclusion. It offers a set of low-level primitives around locks and the high-level `critical` construct to protect the access to shared data (through the ownership of one or more locks). Lock-based mechanisms are complex to use and error prone especially when trying to avoid deadlock situations or when trying to use fine-grain locking to achieve better scalability on highly parallel hardware. Consequently there is currently concern in the programming and computer architecture communities that a parallel programming productivity/performance wall might be looming in the horizon.

Transactional Memory (TM) is a promising mechanism to tackle this problem by abstracting some of the complexities associated with concurrent access to shared data. With TM, multiple threads can simultaneously try to access multiple shared memory locations within the scope of what is called a "transaction". The detection of memory access conflicts causes transactions to rollback.

When compared to TM, locks are pessimistic. With mutual exclusion, only one thread can hold a given lock at a given time whereas with TM more than one thread can access a given critical section simultaneously. Given that actual conflicts are rare in many programs [3], the optimistic TM approach makes much more sense as a future programming model. With TM the programmer specifies intent rather than mechanism (i.e. the programmer can focus on determining where atomicity is necessary, rather than the mechanisms used to enforce it), resulting in a higher-level abstraction than locks.

Figure 1 shows an excerpt from the SpecOMP2001 AMMP program. In this case the programmer uses a lock for each atom (`a1->lock` and `a2->lock`) to protect the access to the fields of the two interacting atoms (`a1` and `a2`). And this is the lock that it is used in each critical region. Due to nature of the program, it is very rare that more than one processor tries to access the critical region with the same lock.

Some recently proposed programming models, such as Sun's Fortress [4], IBMs X10 [5] and Crays Chapel [6], include an atomic statement to define conditional and/or conditional atomic blocks of statements that are executed as transactions. In some cases, atomic can also be an attribute for variables so that any update to them in the code is treated as if the update is in a short atomic section. OpenMP also offers an `atomic` pragma to specify certain indivisible read-operation-write sequences, for which current microprocessor usually provide hardware support. For example through load linked-store conditional (LL-SC).

Two main TM implementation styles stand out: hardware- and software-based. Historically, the earliest design proposals were hardware based. Software Transactional Memory (STM) [7] has been proposed to address, among other

```
#pragma omp parallel default(none)
    shared(dielectric, lambda, a_number, atomall)
    private (imax, i, a1, a2, xt, yt, zt, fx, fy, fz, a1fx, a1fy,
             a1fz, ii, jj, ux, uy, uz, r, k, r0)
{ ... #pragma omp for schedule(guided)
   for( i= 0; i< imax; i++) {
      ...
      a1 = (*atomall)[i];
      ...
      for( ii=0; ii< jj;ii++) {
         a2 = a1->close[ii];
#ifdef _OPENMP
         omp_set_lock(&(a2->lock));
#endif
         ux = (a2->dx -a1->dx)*lambda + (a2->x -a1->x);
         ...
         r =one/( ux*ux + uy*uy + uz*uz);
         r0 = sqrt(r);
         ux = ux*r0;
         ...
         k = -dielectric*a1->q*a2->q*r;
         r = r*r*r;
         k = k + a1->a*a2->a*r*r0*six;
         k = k - a1->b*a2->b*r*r*r0*twelve;
         a1fx = a1fx + ux*k;
         ...
         a2->fx = a2->fx - ux*k;
         ...
#ifdef _OPENMP
         omp_unset_lock(&(a2->lock));
#endif
      }
#ifdef _OPENMP
      omp_set_lock(&(a1->lock));
#endif
      a1->fx += a1fx ;
      a1->fy += a1fy ;
      a1->fz += a1fz ;
#ifdef _OPENMP
      omp_unset_lock(&(a1->lock));
#endif
   }
} /* omp parallel pragma */
```

Fig. 1. Excerpt from the SpecOMP2001 AMMP program

things, some inherent limitations of earlier forms of Hardware Transactional Memory (HTM) [8] such as a lack of commodity hardware with the proposed features, or a limitation to the number of locations that a transaction can access.

Beyond these two main approaches, two additional mixed approaches have recently been considered. Hybrid Transactional Memory (HyTM) [9], [10] supports transactional execution that generally occurs using HTM transaction but which backs off to STM transactions when hardware resources are exceeded. Hardware-assisted STM (HaSTM) combines STM with new architectural support to accelerate parts of the STMs implementation [11], [12]. These designs are both active research topics and provide very different performance characteristics: HyTM provides near-HTM performance for short transactions, but a "performance cliff" when falling back to STM. In contrast, HaSTM may provide performance some way between HTM and STM.

2 Basic Concepts

A transaction is a sequence of instructions, including reads and writes to memory, that either executes completely (commit) or has no effect (abort). When a transaction commits, all its writes are made visible and values can be used by other transactions. When a transaction is aborted, all its speculative writes are discarded.

Commit or abort are decided based on the detection of memory conflicts among parallel transactions. In order to detect and handle these conflicts, each running transaction is typically associated with a 'read set' and a 'write set'. Inside a transaction, the execution of each transactional memory read instruction adds the memory address to the read set. Each transactional memory write instruction adds the memory address and value to the write set of the transaction.

Conflict detection can be either eager or lazy. Eager conflict detection checks every individual read and write to see if there is a conflicting operation in another transaction. Eager conflict detection requires that the read and write sets of a transaction are visible to all the other transactions in the system. On the other hand, with lazy conflict detection a transaction waits until it tries to commit before checking its read and write sets against the write sets of other transactions.

Another fundamental design choice is how to resolve a conflict once it has been detected. Usually, if there is a conflict it is necessary to resolve it by immediately aborting one of the transactions involved in the conflict.

In order to support the execution of a transaction, a data versioning mechanism is needed to record the speculative writes. This speculative state should be discarded on an abort or used to update the global state on a successful commit. The two usual approaches to implement data versioning are based on using an undo-log or using buffered updates. Using an undo-log, a transaction applies updates directly to memory locations while logging the necessary information to undo the updates in case of abort. On the contrary, approaches using buffered updates keep the speculative state in a transaction-private buffer until commit time; if the commit succeeds, the original values before the store instructions are dropped and the speculative stores of the transaction are committed to memory.

HTM proposed systems keep the speculative state of the transactions mostly in the data cache or in a hardware buffer area. Transactional loads and stores can be kept in a separate "transactional cache" or in the conventional data caches, augmented with transactional support. In both cases the modifications are minimal since transactional support relies on extending existing cache coherence protocols such as MESI to detect conflicts and enforce atomicity. On the contrary, STM implementations must provide mechanisms for concurrent transactions to maintain their own views of the heap, allowing a transaction to see its own writes as it continues to run, and allowing memory updates to be discarded if the transaction ultimately aborts. In addition, STM implementations must provide mechanisms for detecting and resolving conflicts between transactions.

3 Our "Proof-of-Concept" Approach

In order to explore how TM could influence the future design and implementation of OpenMP, we have adopted a "proof-of-concept" approach based on a source-to-source code restructuring process, implemented in Mercurium [13], and two libraries to support the OpenMP execution model and to support transactional memory: NthLib [14] and Nebelung [15], respectively. We also propose some OpenMP extensions to specify transactions.

3.1 Is OpenMP ATOMIC a Transaction?

The `atomic` construct in OpenMP ensures that a specific storage location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.

```
#pragma omp atomic
   expression-statement
```

where expression-statement can have a limited number of possibilities, such as `x = x operator expr`. Only the load and store of the object designated by `x` are atomic; the evaluation of `expr` is not atomic (and should not include any reference to `x`).

Some OpenMP compilers just replace atomic regions by critical regions, usually excluding from the critical region the evaluation of `expr`. Others make use of the efficient machine instructions available in current microprocessors to atomically access memory or through the use of load-linked store-conditional. Figure 2 shows that for a simple example: the compiler generates code so that expression on the right is computed first (in this case, a constant value 1). Once evaluated it "saves" the current version of a(i) and applies the operator (+ in this case). Then

```
!$omp parallel
   do i = 1, n
!$omp atomic
      a(i) = a(i) + 1
   enddo
!$omp end parallel
```

(a)

```
atomic_expr_a = 1 atomic_old_a = a(i) atomic_new_a=atomic_old_a+
atomic_expr_a DO WHILE (0 .EQ.
atomic_update_4(a(i),atomic_old_a,atomic_new_a))
   atomic_old_a = a(i)
   atomic_new_a=atomic_old_a+atomic_expr_a
END DO
```

(b)

Fig. 2. Atomic region in OpenMP and a possible translation

it "tries" to update the shared variable, and if it fails, restores the original value of a(i) and starts again applying the operator. The while loop finishes when the `atomic_update_4` function returns successful (different than 0). So notice that this is a simplified version, just for a single variable, of a transaction.

3.2 Is OpenMP CRITICAL a Transaction?

The `critical` construct in OpenMP restricts execution of the associated structured block to a single thread at a time (among all the threads in the program, without regard to the team(s) to which the threads belong).

```
#pragma omp critical [(name)]
   structured-block
```

An optional `name` may be used to identify the `critical` construct. All `critical` constructs without a name are considered to have the same unspecified name. A thread waits at the beginning of a `critical` region until no other thread is executing a `critical` region with the same name. The `critical` construct enforces exclusive access with respect to all `critical` constructs with the same name in all threads, not just in the current team.

OpenMP compilers usually replace critical regions with `set_lock/unset_lock` primitives, declaring a different lock variable for each name used in critical constructs. From this usual implementation and the description above (taken from the current 2.5 language specification) we understand that the code in the structured block can be never speculatively executed by a thread so that several threads are executing it simultaneously.

In this case, a first proposal to integrate TM in OpenMP would be to relax the previous description so that “waiting at the begining of the `critical` region” does not preclude the thread from executing the structured block speculatively as a full transaction. However, we prefered to first propose a set of extensions (construct and clauses) to understant and explore how the TM abstraction could fit with the OpenMP programming style and execution model. We could later see the minimum changes required in the current specification to integrate transactional execution in OpenMP.

3.3 Proposed OpenMP Extensions for TM

The first extension is a pragma to delimit the sequence of instructions that compose a transaction:

```
#pragma omp transaction [exclude(list)|only(list)]
   structured-block
```

With this extension, the programmer should be able to write standard OpenMP programs, but instead of using intrinsic routines to lock/unlock, atomic or critical pragmas, he/she could use this pragma to specify the sequence of statements that need to be executed as a transaction. The optional `exclude` clause can be used to specify the list of variables for which it is not necessary to check for conflicts. This means that the STM library does not need to keep track of them in the read and write sets. On the contrary, if the programmer uses the optional `only` clause, he/she is explicitly specifying the list of variables that need to be tracked. In any case, data versioning for all speculative writes is needed for all shared and private variables in case the transaction needs to be rolled-back.

Another possibility is the use of a new clause associated to the OpenMP worksharing constructs:

```
#pragma omp for transaction [exclude(list)|only(list)]
   for (...;...;...)
      structured-block
```

or

```
#pragma omp sections transaction [exclude(list)|only(list)]
#pragma omp section
   structured-block
#pragma omp section
   structured-block
```

In the first case, each iteration of the loop constitutes a transaction, while in the second case, each section is a transaction.

For OpenMP 3.0, a new tasking execution model is being proposed. 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 worksharing 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 the possibility of tagging a task as a transaction, using the same clause specified above.

```
#pragma omp task transaction [exclude(list)|only(list)]
   structured-block
```

3.4 Nebelung Library Interface and Behavior

In order to have a complete execution environment supporting transactional memory, we have implemented our own STM library, named Nebelung. A detailed explanation of the library and its implementation is out of the scope for this paper; therefore we present the relevant issues here. The library satisfies the interface presented in the Figure 3. Nebelung library is typeless (work on a byte level) so we also developed wrapper functions `read` and `write` around `readtx` and `writetx`, which cast results into the proper types. Note that this interface is similar to the ones provided by other current STM libraries [3].

The library functions have the following semantics: `createtx` and `destroytx` create and destroy the required data structures for the execution of a transaction, `starttx` starts the transaction, `committx` publishes (i.e., makes visible) the results (writes) of the transaction, `aborttx` cancels the transaction and `retrytx` cancels the transaction and restarts it. `readtx` and `writetx` are function library function calls for handling memory accesses. The transaction should be started and ended with the code presented in the Figure 4.

The most important parts of the library are surely function calls `readtx` and `writetx` and they require special attention. Both functions operate on the byte level. `writetx` receives the real address where the data should be stored, a size of data and the data itself. Function `readtx` receives the real address which should be read and the size of the data, and returns the pointer to the

```
Transaction*    createtx    ();
void            starttx     (Transaction *tr);
status          committx    (Transaction *tr);
void            destroytx   (Transaction *tr);
void            aborttx     (Transaction *tr);
void            retrytx     (Transaction *tr);
void*           readtx      (Transaction *tr, void *addr, int blockSize);
void*           writetx     (Transaction *tr, void *addr, void *obj,
                                              int blockSize);
```

Fig. 3. Nebelung library interface to support the code generated by the Mercurium compiler

```
{   Transaction* t = createtx(); while (1) {
   starttx (t);
   if (setjmp (t->context) == TRANSACTION_STARTED) {
```

(a)

```
          if (COMMIT_SUCCESS == committx (t)) break;
          else aborttx (t);
       } else aborttx (t);
   }
   destroytx (t);
}
```

(b)

```
startTransaction();
// transaction body
endTransaction();
```

(c)

Fig. 4. Macros for (a) starting and (b) ending a transaction. (c) Code of the transaction surrounded by the previous macros.

location which holds the requested data. Returned pointer does not need to be the same as the original one and this is implementation dependent. This can be the source of a memory leakage, because it is not clear who should release the pointed memory. There are two possible implementations of function `readtx`. The first implementation option can copy the data to a new location and return the pointer, requiring the programmer to free it when it is not needed any more. The other option, which we implemented, is that the library takes care about everything and returns the pointer to the location which should be just read. This pointer should not be used later for writing. If the same data should be modified later, that should be done through function `writetx` and not directly through returned pointer.

The current implementation of Nebelung library performs lazy conflict detection. Read and write sets are maintained dynamically and all memory operations are performed locally for the transaction. At commit time, the library checks if there is any conflict with other transactions. If conflict exists, the current transaction is committed and other transactions are aborted. In this way the transaction progress is guaranteed.

3.5 Source-to-Source Translation in Mercurium

The Mercurium OpenMP source-to-source translator transforms the code inside the `transaction` block in such a way that for each memory access, a proper

```
#pragma omp transaction {
   ux = (a2->dx -a1->dx)*lambda + (a2->x -a1->x);
   r =one/( ux*ux + uy*uy + uz*uz);
   r0 = sqrt(r);
   ux = ux*r0;
   k = -dielectric*a1->q*a2->q*r;
   r = r*r*r;
   k = k + a1->a*a2->a*r*r0*six;
   k = k - a1->b*a2->b*r*r*r0*twelve;
   a1fx = a1fx + ux*k;
   a2->fx = a2->fx - ux*k;
}
```

(a)

```
{ startTransaction(); {write(t, &ux, (*read(t, &((*read(t,
&a2))->dx))-
    *read(t, &( ( *read(t, &a1))->dx))) *
    *read(t,&lambda)+(*read(t,&((*read(t,&a2))->x))-
    *read(t, &( ( *read(t, &a1)) ->x))));
 write(t, &r,  *read(t, &one) /( *read(t, &ux) *
    *read(t, &ux) + *read(t, &uy) * *read(t, &uy) +
    *read(t, &uz) * *read(t, &uz) ));
 write(t, &r0, sqrt( *read(t, &r) ));
 write(t, &ux,  *read(t, &ux) * *read(t, &r0) );
 write(t, &k, - *read(t, &dielectric) *
    *read(t, &( ( *read(t, &a1)  ) ->q) ) *
    *read(t, &( ( *read(t, &a2)  ) ->q) ) *
    *read(t, &r) );
 write(t, &r,  *read(t, &r) * *read(t, &r) *
    *read(t, &r) );
 write(t, &k,  *read(t, &k) +
    *read(t, &( ( *read(t, &a1))->a) ) *
    *read(t, &( ( *read(t, &a2))->a) ) *
    *read(t, &r)* *read(t, &r0) * *read(t, &six));
 write(t, &k,  *read(t, &k)
    *read(t, &((*read(t, &a1))->b) ) *
    *read(t, &((*read(t, &a2))->b) ) *
    *read(t, &r) * *read(t, &r) * *read(t, &r0) *
    *read(t, &twelve));
 write(t, &a1fx,  *read(t, &a1fx)+ *read(t, &ux)* *read(t, &k) );
 write(t, &( ( *read(t, &a2)  ) ->fx ) ,
    *read(t, &( ( *read(t, &a2)  ) ->fx) )
    *read(t, &ux) * *read(t, &k) );
} endTransaction(); }
```

(b)

Fig. 5. Code generated code for the first critical section in Figure 1 (excerpt from AMMP SpecOMP2001)

```
int f(int); int correct(int* a, int* b, int* x){
   int fx;
#pragma omp transaction exclude (fx) {
   fx = f(*x);
   a += fx;
   b -= fx;
   }
}
```

(original code)

```
int correct(int* a, int* b, int* x){
   int fx;
   { startTransaction();
      {
      fx = f(*read(t, x));
      write(t, &a, *read(t, &a) + fx);
      write(t, &b, *read(t, &b) - fx);
      }
   endTransaction();
   }
}
```

(transactional code)

Fig. 6. Example using the exclude clause

STM library function call is invoked. The current version of Mercurium accepts OpenMP 2.5 for Fortran90 and C.

Figure 5 shows how the first critical region in Figure 1 is specified using our proposed extensions and the code generated by Mercurium. Figure 6 shows a synthetic example using the `exclude` clause.

Finally, Figure 7 shows another example which operates on a binary tree, inserting n nodes into the tree. We are using the current tasking proposal for OpenMP 3.0. Notice that n tasks will be created and executed atomically in parallel. Figure 8 shows the code generated by Mercurium.

3.6 Support for Hardware Transactional Memory

Researchers have not yet built a conscesnus about the best degree of harware support for transactional memory (i.e. full HTM, HaSTM or HyTM), particularly when looking at future multicore architectures with large numbers of cores on a chip and not necessarely cache coherent. This is part of our current work and orthogonal to the transformation process discussed in this paper, which can easily be retargeted to support HTM: we only need to change the `startTransaction` and the `endTransaction` macros. Those macros should use the proper hardware

```
ivoid ParallelInsert(struct BTNode** rootp, int n, int keys[], int
values[]){ #pragma omp parallel single {
   for (int i = 0; i < n; ++i) {
      int key = keys[i], value=values[i];
#pragma omp task capturevalue(key, value) captureaddress(rootp) {
      int inserted, f;
      BTNode* curr;
      BTNode* n;

      n = NewBTNode;
      initNode(n,key,value);
      inserted = 0;
      f = 0;
#pragma omp transaction {
      if (*rootp == 0) {*rootp = n;}
      else {
         curr = *rootp;
         while (inserted == 0) {
            if (curr->key == key){
               curr->value = value;
               curr->valid = 1;
               inserted = 1;
               f = 1;
            } else if (curr->key> key) {
               if (curr->left == 0) {curr->left = n; inserted = 1;}
               else curr = curr->left;
            } else {
               if (curr->right == 0) {
                  curr->right = n;
                  inserted = 1;
               } else curr = curr->right;
            }
         }
      }
      } // end oftransaction
      if (f == 1) free(n);
      } // end of task
   } // end if for loop
   } // end of parallel region
} // end of ParallelInsert function
```

Fig. 7. Function for parallel insertion of n nodes into a binary search tree, expressed using the tasking execution model

ISA instructions for the start and the end of the transaction. The transformation of loads and stores is not needed anymore. To illustrate this, in this paper we chose to incorporate the HTM proposed by McDonald et. al [16] with instructions

```
{ startTransaction(); { if (  *read(t, rootp)  == 0 ) { write(t,
rootp,  *read(t, &n) );
  } else {
     write(t, &curr,  *read(t, rootp));
     while (  *read(t, &inserted)  == 0) {
        if (*read(t, &((*read(t,&curr))->key))== *read(t, &key)) {
            write(t, &((*read(t, &curr))->value), *read(t, &value));
            write(t,&((*read(t,&curr))->valid),1);
            write(t, &inserted, 1);write(t, &f, 1);
        } else if (*read(t,&((*read(t, &curr))->key)) > *read(t, &key)) {
            if(*read(t,&((*read(t, &curr))->left)) == 0 ) {
               write(t, &((*read(t, &curr))->left), *read(t, &n) );
               write(t, &inserted, 1);
           } else
               write(t, &curr, *read(t,&((*read(t,&curr))->left)) );
        } else { if(*read(t,&((*read(t,&curr))->right)) == 0 ) {
               write(t,&((*read(t, &curr))->right ), *read(t, &n) );
               write(t, &inserted, 1);
           } else write(t, &curr, *read(t, &((*read(t,&curr))->right)));
        }
      }
   }
} endTransaction();}
```

Fig. 8. Code generated for the transaction inside the parallelInsert function

```
#define startTransaction() \        #define endTransaction()     \
  { asm { xbegin }                      asm { xvalidate          \
                                              xcommit            \
                                            }                    \
                                      }
```

Fig. 9. Support for HTM. Definition in pseudo code, of `startTransaction` and `endTransaction` macros in case hardware has instructions `xbegin` and `xcommit` for the start and the end of the transaction, and `xvalidate` for the validation of the transaction.

`xbegin, xcommit` and `xvalidate` (denoting the start, end and validation of the transaction, respectively). In this case, the start and end macros would be as shown in Figure 9. Note that `endTransaction` semantics require `xvalidate` since [16] uses two-phase commit.

For those variables that do not need to be tracked, the HTM proposed in [16] includes instructions `imld` and `imst` for load and store without changing read and write sets. Figure 10 shows how the compiler would generate code for a simple `a = b + c` statement, where `c` does not need to be tracked, for both STM and HTM.

4 Open Issues

In addition to the HTM implementation challenge mentioned in the previous section, there are other issues that become important challenges when TM is incorporated into OpenMP. Some issues are research challenges while other are restrictions on the way OpenMP and TM can be combined.

The first issue refers to transaction nesting. Nested transactions can be supported in two different ways: closed nested or open nested. In a closed nested TM system either all the transactions that are in a nested region commits or neither. In comparison, in an open nested TM when an inner transaction commits its effects are made visible for all threads in the system. The use of open nested transactions can unleash more concurrency than closed nested ones. When an open nested transaction commits, its write set is made visible to all other transactions, so other transactions can see the modifications sooner and work with this modified data. However open nested transactions increase the burden of the programmer: compensating actions are needed when the outermost transaction commits and when one of the surrounding transactions aborts. The handling of this compensating code could be quite complex, and the programmer must have an expert level grasp of the semantics of the code. For this reason, although OpenMP-TM can incorporate closed-nested transactions relatively easily, open-nested transactions are challenging because the compensating code could impact the sequential nature of the program when it is compiled without OpenMP.

```
a = b + c;        write(t, &a, *read(&b) + c);       asm {
                                                         load  r0, b
                                                         imld  r1, c
                                                         add   r0, r1
                                                         store a,  r0
                                                         }


(original code)   (transformed STM code)             (transformed HTM code)
```

Fig. 10. Transformation of a simple statement using ii) STM or iii) HTM in pseudo assembler code

The second issue refers to the use of I/O inside a transaction. I/O inside a critical section is not a problem for OpenMP because such blocks are protected and never rolled back. However, for TM the issue is different: suppose that inside a transaction, a system call attempts to output a character to the terminal. One solution is to execute the system call immediately; however it would be very problematic if this transaction aborts later. Trying to “undo” the I/O by deleting the character upon an abort would obviously lead to a very wobbly system. In some cases, even executing the I/O operation may be difficult if the data is buffered in HTM. A different approach to solve the I/O problem would be to categorize I/O based on their abortive properties. By definition, an I/O call is undoable if its effects could be rolled back which in turn implies that its effects are self-contained to the I/O operation only. Here, the challenge is to allow maximum programmer expressiveness while avoiding too complex implementations. Inevitably, programmers must be aware of certain kinds of I/O operation that simply do not make sense to transparently perform as part of a single atomic transaction: for instance, prompting the user for input and then receiving and acting on the input. However note that those compensating actions can also affect the sequential nature of the program when it is compiled without OpenMP. We believe that a first-order approach is to forbid the I/O operations in OpenMP-TM transactions.

Another issue is about the nesting of OpenMP constructs and transactions. There are two cases that one needs to consider. In the first case, a transaction might exist inside an OpenMP parallel region or worksharing construct. Note that the transaction might be invoked from within a library call, so the programmer might not be aware of this. However, even with this added complexity, this case is easy to handle: each thread in the team will be having a transaction inside. The other case, where an OpenMP parallel or worksharing construct appears inside the scope of a transaction, is more complicated. Note that for this case as well, the programmer may not be aware that an OpenMP construct exists within the transaction; this can happen if the programmer uses library function inplemented in OpenMP. The complexity rises from the fact any of the OpenMP threads are liable to be aborted at any time. This signifies that if one created thread is aborted, then all the other threads must also abort unnecessarily due to the semantics of TM. This can be avoided using a mechanism like close-nested transaction allowing to rollback only the conflicting thread if it is possible to ensure that the conflict only affects this thread.

Finally, it would be interesting to include additional functionalities to the basic transactional execution model offered by `transaction`. For example, we could specify a condition; if the evaluation of the condition returns false then the transaction is known to abort. Similarly, we could add a condition that needs to be evaluated once a transaction is aborted; if the evaluation of the condition returns false, it is known that a reexecution of the transaction will fail again. So it is better no to execute it until the condition is true. These conditions could for example include operators to check that a shared variable has been accessed (`touch(var_name)`).

```
#pragma omp transaction [retrywhen(condition)][executeif(condition)]
   structured-block
```

5 Conclusions

In this paper we have done a first exploration of how Transactional Memory (TM) could provide a more graceful and natural mechanism to write parallel programs than using lock-based mechanisms. Recently there is a flurry of research activity to define and design Hardware and Software Transactional Memory, and our OpenMP community should follow this activity in order to take benefit as soon as possible. This paper has covered a basic proposal to extend OpenMP with transactions and identified some significant future challenges in the OpenMP-TM tandem. However, the biggest challenge is to make the adoption of Transactional Memory by the (OpenMP) programmer community as smooth as possible through HW/SW design; effective mechanisms for supporting TM are crucial to fulfilling the promise of improved application performance on future many-core CMPs.

During the preparation of this version of the paper, we realized that a paper on a similar topic was accepted for publication [17]. However, only the abstract of the other paper was available to us, so we were not able to make any comparisons at this point.

Acknowledgments. This work is supported by the cooperation agreement between the Barcelona Supercomputing Center National Supercomputer Facility and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2004-07739-C02-01 and by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC).

References

1. Larus, J., Rajwar, R.: Transactional Memory. Morgan Claypool, San Francisco (2006)
2. OpenMP Architecture Review Board, OpenMP Application Program Interface (May 2005)
3. Harris, T., Plesko, M., Shinnar, A., Tarditi, D.: Optimizing Memory Transactions. In: PLDI 2006. ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (June 2006)
4. Allen, E., Chase, D., Luchangco, V., Maessen, J.-W., Ryu, S., Steele Jr., G.L., Tobin-Hochstadt, S.: The Fortress Language Specification. Sun Microsystems (2005)
5. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an Object-oriented approach to non-uniform Cluster Computing. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA), New York, USA, pp. 519–538 (2005)
6. Cray. Chapel Specification (February 2005)
7. Shavit, N., Touitou, D.: Software Transactional Memory. In: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, pp. 204–213 (1995)
8. Herlihy, M., Eliot, J., Moss, B.: Transactional Memory: Architectural Support for Lock-Free Data Structures. In: Proc. of the 20th Int'l Symp. on Computer Architecture (ISCA 1993), May 1993, pp. 289–300 (1993)
9. Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D.: Hybid Transactional Memory. In: Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (October 2006)

10. Kumar, S., Chu, M., Hughes, C.J., Kundu, P., Nguyen, A.: Hybrid Transactional Memory. In: Proceedings of ACM Symp. on Principles and Practice of Parallel Programming (March 2006)
11. Saha, B., Adl-Tabatabai, A., Jacobson, Q.: Architectural Support for Software Transactional Memory. In: 39th International Symposium on Microarchitecture (MICRO) (2006)
12. Shriraman, A., Marathe, V.J., Dwarkadas, S., Scott, M.L., Eisenstat, D., Heriot, C., Scherer III, W.N., Spear, M.F.: Hardware Acceleration of Software Transactional Memory. In: TRANSACT 2006 (2006)
13. Balart, J., Duran, A., Gonzàlez, M., Martorell, X., Ayguadé, E., Labarta, J.: Nanos Mercurium: A Research Compiler for OpenMP. In: European Workshop on OpenMP (EWOMP 2004), Stockholm, Sweden, October 2004, pp. 103–109 (2004)
14. Martorell, X., Ayguadé, E., Navarro, N., Corbalan, J., Gonzalez, M., Labarta, J.: Thread Fork/join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors. In: 13th International Conference on Supercomputing (ICS 1999), Rhodes (Greece) (June 1999)
15. Milovanović, M., Unsal, O.S., Cristal, A., Stipić, S., Zyulkyarov, F., Valero, M.: Compile time support for using Transactional Memory in C/C++ applications. In: 11th Annual Workshop on the Interaction between Compilers and Computer Architecture INTERACT-11, Phoenix, Arizona (February 2007)
16. McDonald, A., Chung, J., Carlstrom, B., Minh, C., Chafi, H., Kozyrakis, C., Olukotun, K.: Architectural Semantics for Practical Transactional Memory. In: Proc. 33th Annu. international symposium on Computer Architecture, pp. 53–65 (2006)
17. Baek, W., Minh, C.-C., Trautmann, M., Kozyrakis, C., Olukotun, K.: The OpenTM Transactional Application Programming Interface. In: Proc. 16th International Conference on Parallel Architectures and Compilation Techniques (PACT 2007), Romania (September 2007)