Practical Experience with Nebelung: The Runtime Support for Transactional Memory and OpenMP

Miloš Milovanović, Roger Ferrer, Vladimir Gajinov, Osman S. Unsal,
Adrian Cristal, Eduard Ayguadé and Mateo Valero

***Abstract*— Transactional Memory (TM) is a key future technology for emerging many-cores. On the other hand, OpenMP provides a vast established base for writing parallel programs, especially for scientific applications. Combining TM with OpenMP provides a rich, enhanced programming environment and an attractive solution to the many-core software productivity problem.**

In this paper, we discuss the runtime environment for supporting our combined TM and OpenMP framework. Through motivating examples, we briefly illustrate how the combined TM/OpenMP can facilitate writing parallel programs. We then discuss issues in runtime environment design. In another contribution, we introduce a set of applications that we specifically developed for our combined TM/OpenMP framework. Using those applications, we show how we can use the rich features provided by TM such as retry to support efficient TM/OpenMP programs. We also include an initial performance analysis of the runtime using the applications.

***Index Terms*— OpenMP, Transactional Memory, Runtime Environment**

I. Introduction

The advent of shared-memory multi-core microprocessors has created an opening to exploit thread-level parallelism. In most applications, parallel thread execution requires synchronization or ordering mechanisms for accessing shared data. Traditional multithreaded programming models usually offer a set of low-level primitives, such as locks, to guarantee mutual exclusion; ownership of one or more locks protects access to shared data. However, locks are complex to use and error prone—especially when a programmer is trying to avoid deadlock situations or to achieve better scalability on highly parallel hardware by using fine-grained locking. Consequently, the programming and computer architecture communities are concerned that a parallel-programming productivity and performance wall might be looming. 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. TM makes it relatively easy to develop parallel programs.

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. A significant number of applications have been parallelized for shared memory multiprocessor systems using OpenMP. Consequently, a large programmer base exists with significant OpenMP expertise. OpenMP 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).

We believe that the fusion of OpenMP with TM offers the best of both worlds for developing parallel programs for many-cores: an established programmer base coupled with a more intuitive way of writing shared-state programs.

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.

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 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 STM's 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.

Some recently proposed programming models, such as Sun's Fortress [4], IBM's X10 [5] and Cray's Chapel [6], include an atomic statement to define atomic 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 microprocessors offer this support through load linked-store conditional (LL-SC). However, note that the atomic `pragma` is extremely limited in this case. In our previous work [18], we described how OpenMP can be combined with TM to offer a fully-featured atomic program section.

In this paper, we introduce the Nebelung runtime which brings together TM with OpenMP and which can be easily customized to work with STM, HTM, HyTM or HaSTM systems. To keep the text brief, here we consider an STM-based TM implementation. In parallel with this work, Baek et al. proposed the The OpenTM Transactional Application Programming Interface [17]. Their proposal is similar to our OpenMP-TM extension [18] with additional constructs for features such as conditional synchronization, nesting, transactional handlers, and contention management. Moreover, OpenTM has been implemented using a compiler-based approach as opposed to our source to source compiler. Nevertheless, it is reasonable to merge the two proposals into a unified set of OpenMP extensions for TM programming.

Contributions of our paper are:

- We present the first multithreaded STM runtime with eager conflict detection in a separate thread.
- We present the retry feature in real applications
- We present specific applications written using the combined OpenMP, TM framework
- We dwell on the problem of "notifyAll" overhead, which occurs when using retry. We proposed the concept of the transaction queues to solve it.

II. 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.

III. 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 the new OpenMP extensions to specify transactions.

A. Proposed OpenMP extension 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 do the rolled-back.

Another possibility is the use of a new clause associated to the OpenMP work sharing 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 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 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
```

B. Nebelung library interface and behavior

In order to have a complete execution environment supporting transactional memory, we have implemented our own STM library and runtime system, named Nebelung. The library satisfies the interface presented in the Figure 1. 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].

```
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);
void*         invalidateAddr(Transaction* tr,
                             void* addr);
void*         invalidateRange(Transaction* tr,
                void* fromAddr, void* toAddr);
status        validatetx(Transaction *tr);
status        resolveConflicttx(Transaction *t);
void          _mCommittx(Transaction *tr);
```

Fig.1. Nebelung library interface. The interface provides functions for maintenance of the memory transactions. It is very generic so behind that interface can be any implementation of TM: STM, HyTM or HaSTM.

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 functions for handling memory accesses. The transaction should be started and finished with the code presented in the Figure 2.

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 location which holds the requested data. Returned pointer does not need to be the same as the original one and this is implementation dependent. Returned value should be just read only once, after that it is invalidated and cannot be used anymore. Writes need to be done through `writetx` function and not through the pointer returned with `readtx`. Responsibility of those functions it so notify the runtime that transaction is trying to access to some memory location.

```
{    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. 2. Macros for (a) starting and (b) ending a transaction. (c) Code of the transaction surrounded by the previous macros.

Functions `invalidateAddr` and `invalidateRange` are used for dealing with the stack variables. We are doing the source to source translation so we don't have directly under our control operations on stack variables (e.g. placing and removing of function arguments). So we need to invalidate stack variables after the return from the function which was using those variables. Although functions `invalidateAddr` and `invalidateRange` were introduced to solve the technical problems they are also useful for potential implementation of early release [19] [21] in OpenMP-TM. For example, Unread-Tvar in Haskell is a transactional variable which can be excluded from the read set before the end of the transaction thus providing early release (example: in case of insertion in a sorted linked list we can remove list elements from the read set when the transaction passes them, because their change will not affect the result of the transaction).

Functions `validatetx`, `resolveConflicttx`, and `_mCommittx` are obtained by decomposition of the function `committx`. Those functions are used for conflict detection and resolution and for the publishing of the temporary transactional data respectively. We identified those functions because we noticed that they can be parallelized and executed in pipeline fashion in some cases. Function `validatetx` can be perfectly parallelized because conflict checking with multiple

transactions is inherently parallel. As it will be described in the next section those functions can be executed in different threads of the Nebelung runtime for performance gains.

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.

C. *Nebelung Runtime System*

Nebelung runtime system is responsible for the management of the memory transactions. In the beginning we implemented a static STM library but after the initial tests we realized that many tasks in the STM library can be done in parallel with the execution of the transactions. A typical example is the eager conflict detection which can be done by a separate thread on a dedicated core, completely in parallel with the transactional execution. Because very soon we will have hundreds of cores on a single die, we can use some of them just for the runtime, which can also scale according to the active workload. Figure 3 shows the overheads which exist in the static STM systems.

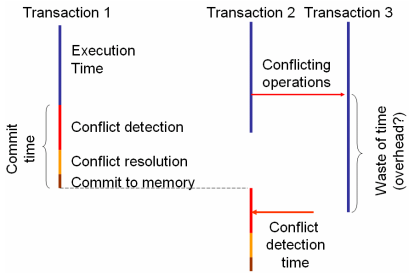


Fig. 3. Static STM Library overheads and the expected gain in case of the active runtime with the eager conflict detection. Two mayor overheads are the commit time which directly increase the total transaction time and the time which transactions spends in the wasteful execution until the conflict is detected.

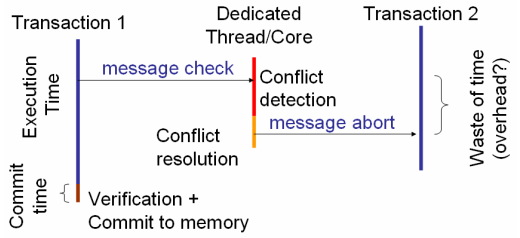


Fig. 4. Expected time diagram with the active STM runtime system. Transactions just sent the message to the runtime when they will access to some memory location. Then runtime will check if there is a conflict with other transactions, in case of conflict resolve it and send a message to the conflicting transaction to abort.

We designed the Nebelung runtime system so that it is executed in separate threads from transactions. As soon as some transaction tries to access some memory location it will just send a message to the runtime and continue the execution. Runtime system is responsible for the transaction management. Among other operations, the runtime will perform conflict detection and resolution. As soon as the runtime detects and resolves the conflict it will notify the conflicting transaction to abort. Figure 4 shows the expected behavior of Nebelung runtime.

Runtime system should lower the two overheads in the transactional system: 1. conflict detection and resolution will be done in parallel with the transaction execution so it will not increase the transaction execution time and 2. the conflicting transaction will be notified to abort immediately when the conflict appears so that transaction will not waste the time on the calculation with the incorrect data. Complete implementation of the Nebelung runtime is the part of our future work.

In this paper we present the preliminary results of this idea which are promising in Section IV-D. We implemented the system with the dedicated thread just for conflict detection. Transactions only send a message to this thread, when they access to some memory location, and a separate thread is checking for conflicts. In case the conflict is detected, the conflicting transaction which started the execution later will be aborted. Transactions occasionally execute the call to the runtime function `validatetx` to check if they are still valid and can proceed with the execution.

D. *Transactional Memory Feature: Retry*

Retry is a very useful language feature which was proposed for Transactional Memory [3]. The retry implementation allows transactional execution to continue speculatively as long as possible and if some condition is not satisfied the transaction will be re-executed from the beginning. Most importantly, retry semantics require that the transaction should be re-executed if and only if its read set changes (if transaction is re-executed immediately the result will be probably the same with the same read set so transaction will busy wait). Figure 5a shows the syntax of the retry directive and figure 5b shows the usage of the retry for the implementation of the bounded buffer.

```
#pragma omp retry [trigger (set)][when (contidion)]
```

Fig. 5a. Proposal for the OpenMP retry directive. Retry can be used without any option and transaction will wait until anything in its read set is changed. If the trigger set is given then transaction should be restarted when something is changed in the given trigger set. If `when` condition is set then transaction should be restarted when the condition is satisfied. A similar concept to the trigger set is proposed by McDonald et al. [16].

```
int putBB(BBuffer* buf, int val){
#pragma omp transaction
{
  while (buf->size == buf->cap)
    #pragma omp retry
    ;
  buf->size++;
  buf->a[buf->i] = val;
  buf->i = (buf->i + 1) % buf->cap;
}
}
```

```
int getBB(BBuffer* buf){
  int r;
#pragma omp transaction
{
  while (buf->size == 0)
    #pragma omp retry
    ;
  buf->size--;
  r = buf->a[buf->j];
  buf->j = (buf->j + 1) % buf->cap;
}
  return r;
}
```

Fig. 5b. Bounded buffer implemented using Transactional Memory and its retry feature. Code is simple as with coarse grain lock-base approach and fine grain performance will be gained by the runtime. If buffer is full in the moment of insertion or empty in case of removal proper transaction will be blocked until the buffer is changed. While loops are needed in order to follow the semantics of OpenMP that pragmas can be removed and program need to be correct. Without the retry transaction will spin until it is aborted.

We can see from the figure 5 that the retry implementation in OpenMP–TM is a very user friendly feature: the programmer doesn't need to worry about the conditional synchronization or locks. He or she should just wait until the data is changed. In this way we have implemented data flow programming model using transactional memory. The transaction will be executed atomically when its data is ready. If we chain the transactions properly, when one transaction commits, it can trigger the other one and so on. When transaction commits it will do retry again and wait for the new input. In this way we provided the data flow between the transactions. This is the easy programming model with good performance for free.

Retry feature can be used after the conflict resolution. Instead to abort one of the conflicting transaction, transaction should do retry. This approach has the two advantages. First, transaction which is "aborted" will be blocked until the other transaction from the conflict commits. Gain of this approach is that conflicting transaction will not start again until the other transaction finishes. In that way the new conflict will be avoided. Second, retry is much faster for implementation then abort. In general abort means destroy (free) the transaction which can take the significant time and retry means just invalidate temporary data and start again.

E. Contention management

Previous work confirmed that Transactional Memory performs poorly when the rollback rate is high. This is natural because the TM is expected to be used in cases when the conflicts are rare. But, in any case we need to handle the applications and the application phases when the conflict rate is high. To do this we added two functions in our runtime: `entryPolicy(Transaction* t)` and `exitPolicy(Transaction* t)`, which are called when transaction should be started or aborted/committed respectively. They are the interface to the contention manager which is the subject of our current work. The idea of the contention manager is that in cases of the high contention execution of the newly created transactions are stopped. When contention manager detects the high contention it should postpone the execution of the newly created transaction or event and it can even abort some already running transactions. In this way we are lowering the conflict rate and increasing the throughput.

F. Parallel region and Transactions

In our current proposal and the implementations of the OpenMP library and the STM Nebelung Runtime we allowed the transactions inside the OpenMP parallel region. Currently we are working on a support for parallel regions inside the transactions. This will allow that transaction can span over more than one thread. That can be very useful for example in case of the atomic matrix multiplication.

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

Figure 6 shows the representative part of the AMMP Spec OMP 2001 application which should be executed inside the transaction and the code generated by Mercurium. Figure 7 shows a synthetic example using the exclude clause.

Finally, Figure 8 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 9 shows the code generated by Mercurium.

```
#pragma omp transaction {
   …
   r0 = sqrt(r);
   ux = ux*r0;
   k = -dielectric*a1->q*a2->q*r;
   ...
}
```

(a)

```
{ startTransaction();
…
 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) );
 …
endTransaction(); }
```

(b)

Fig. 6. Excerpt from AMMP SpecOMP2001. The code which should be executed atomically (a) and the generated code(b).

```
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();
   }
}
```

(transformed code)

Fig. 7. Example using the exclude clause.

```
void 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;
            …
      }
      } // end oftransaction
      if (f == 1) free(n);
      } // end of task
   } // end if for loop
   } // end of parallel region
} // end of ParallelInsert function
```

Fig. 8. Function for parallel insertion of n nodes into a binary search tree, expressed using the tasking execution model.

```
{ 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));
  …
}
endTransaction();}
```

Fig. 9. Code generated for the transaction inside the `parallelInsert` function.

IV. Results

We tested our system for performance with the tree applications: B+tree, Gauss-Seidel finite difference method application and the Producer/Consumer application. We executed those applications on an 8 core Intel(R) Xeon™ CPU machine on a 3.2GHz and the 8MB of cache. Subsections *A*, *B* and *C* presents the results without the dedicated thread for the conflict detection and subsection *D* compares the execution of the test application on a runtime with and without the dedicated thread.

A. B+tree application

In B+tree application we tested the performance of our system when it operates on a complex data structure with huge amount of data (~100MB). We executed 80 transactions per thread where each transaction atomically takes the data from one B+tree structure, performs compute-intensive non conflicting operations for an average of 50µs and stores the result in the other B+tree.

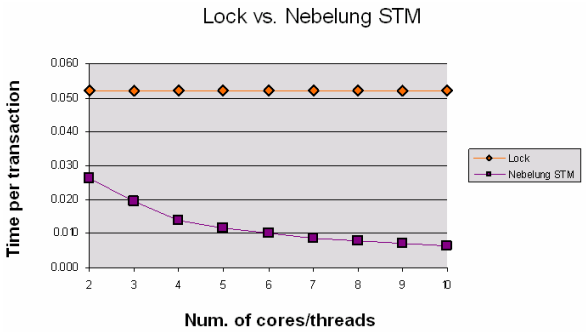


Fig. 10. Results for the B+tree application. Test compares the average transaction execution time of the Nebelung Runtime system and the appropriate coarse-grain lock based system. Results show performance on a par with a fine-grain lock based approach even though the programming model is as easy as a coarse-grain lock based approach.

Figure 10 compares the average transaction time using coarse grain lock-based approach and Nebelung TM system. TM is better because B+tree structures are filled with the huge amount of data so the probability of the conflict is very low (read sets intersects but write sets and read sets don't) so transactions are really executed in parallel without too many aborts. This is obvious for example in case with 2 cores, TM approach is two times faster then the coarse grain lock-based approach.

In case of the B+tree data structure performance of the coarse-grain and fine-grain lock based solution is almost the same. That is because when we insert the new value in a B+tree there is a possibility that even root node can be changed so we need to lock the root also and that is exactly what we are doing in case of coarse-grain solution. So with this application we showed that our system and Transactional Memory in general can give better performance in cases when fine-grain solution doesn't exist or is to difficult to be implemented (e.g. two phase insert in B+tree or similar).

B. Gauss-Seidel application

In Gauss-Seidel application we implemented Gauss-Seidel finite difference method for solving the linear system of equations [20] using Transactional Memory and retry. For each element of the matrix we created a separate thread, which will iteratively calculate the value of that element. Element $m^T_{i,j}$ (value of the element at the position *(i, j)* in the iteration *T*) should be calculated using the expression:

$$m^T_{i,j} = f(m^T_{i-1,j}, m^{T-1}_{i+1,j}, m^T_{i,j-1}, m^{T-1}_{i,j+1})$$

As we can see from the above expression, in order to calculate the element at position *(i, j)* at iteration *T*, its adjacent elements needs to be in the proper iteration (data dependency). That is where we used the retry to create a nice parallel

application, without any problem with synchronization and conditional variables and locks. Figure 11 presents the part of the code of the Gauss-Saidel application and figure 12 presents the obtained results.

```
#pragma omp transaction
{
  x1=0;
  if (r-1>=0){
    while (m[r-1][c].step != t)
      #pragma omp retry

    x1=m[r-1][c].value;
  }
                ...
}
```

Fig. 11. Sample of the code from Gauss-Seidel application. In order to calculate element at position *(i, j)* in the iteration *T*, we need to wait for its adjacent elements to be in the proper iteration. That is where retry can perfectly help.

Results obtained from Gauss-Seidel application shows that systems scales very well until the square matrix dimension 6, because number of threads raises with the square of the matrix dimension. In that moment number of threads/transactions is 36 and the number of available cores is 8 so effect of trashing begin to be dominant. But with the larger number of cores we expect that our application will continue to scale even for the bigger matrices.

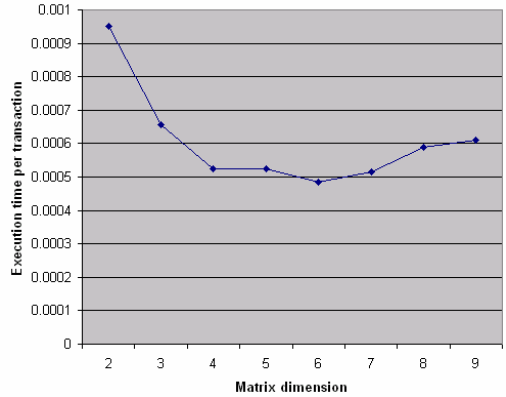
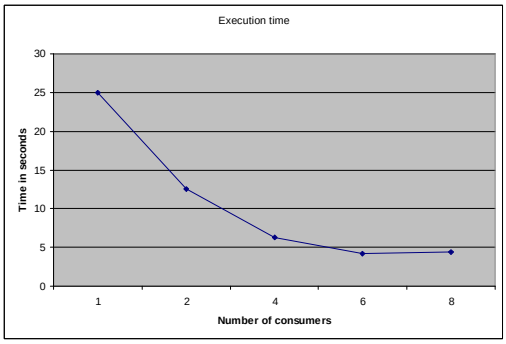


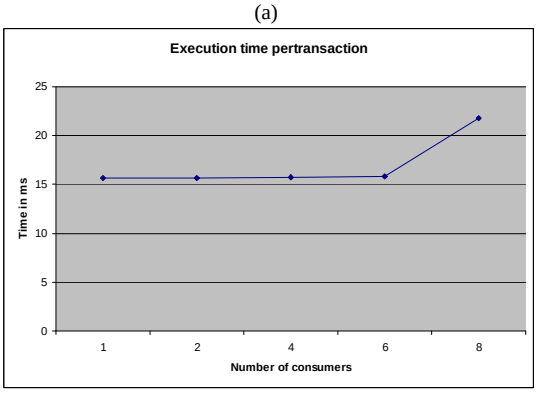
Fig. 12. Results achieved from Gauss-Seidel application. Number of threads is equals to the square of the matrix dimension and each thread is executing one transaction at the time.

C. *Producer/Consumer application*

In the Producer/Consumer application we tested the performance of the bounded buffer we shown in the figure 5b. We created the application with one producer and many consumers. We set that producer is eight time faster than consumers and varied the number of consumer for the constant workload produced by the producer. Results are presented in the figures 13a and 13b.



(a)



(b)

Fig. 13. Results for the Producer/Consumer application. Figure a) presents the total execution time and figure b) presents the average execution time per transaction. Total time scales well with the increased number of cores but the average transaction time is a little bit higher because of “notify all overhead”.

Results show that our system scales very well and that proposed retry feature have very big potential. Retry gives the programmer user friendly programming model from one side and good performance for free. With this test we also discovered the overhead we named “notify all overhead”. It is very similar with the `notifyAll` feature which exists in Java for notification to all threads which are synchronized on the same object. We noticed that average transactional execution increases because when the producer stores the data in the buffer all clients are notified and of course only one will succeed and all others will be aborted. Figure 13b shows that “notify all” overhead can rise with the number of cores. In our future work we will pay more attention to lower this overhead. Currently we propose something like the retrying transaction queue. Idea is that when the queue is notified, it will select the one transaction from the queue and notify it. Other will remain to do retry without wakeup. This can be good idea in any client/server application when we have more servers of the same type so it is not important which will serve the request. It is just important that the proper server queue is notified, and that the queue delivers the request to the proper server (e.g. load balancing).

D. *Dedicated Conflict Detection Thread Results*

This section presents the results of the runtime system with the dedicated conflict detection thread (CDT). Figure 14 presents the results of the application which has *n* dependant transactions and *m* independent transactions executed on a 2 core machine. Results with the CDT are much better for two reasons: 1. CDT performs the eager conflict detection and 2.

there is no overhead for eager conflict detection in the running transactions because it is performed in the dedicated thread. The only overhead for this kind of system is overhead for sending the message to the CDT. This issue presents interesting tradeoffs and will be addressed in future work.

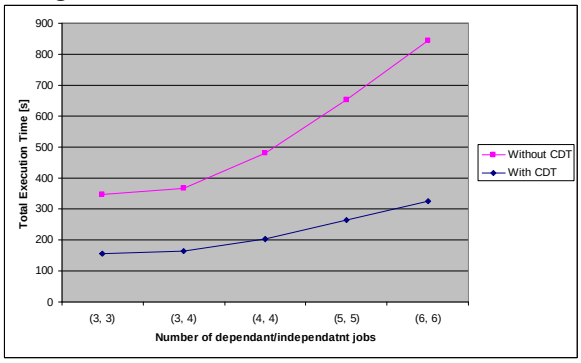


Fig. 14. Results of the test application executed on Nebelung runtime with and without the dedicated core for the conflict detection. (*n*, *m*) stands for the application which has *n* dependant transactions and *m* independent transactions. System with the CDT is scaling much better.

V. Conclusion

In a previous paper we presented the language design for integrating OpenMP with TM, presented solutions for some challenges and dwelled on some open issues. In this paper, we describe our runtime environment, Nebelung, for supporting OpenMP-TM. We also introduced new ideas made possible by the combined OpenMP-TM framework. One such idea is to use separate runtime threads for Conflict Detection which are then executed on available idle cores; we include a first-order feasibility study to confirm the promise of the idea. Future work consists of significantly expanding on this idea, including a full implementation that will be supported by the Nebelung runtime.

VI. Acknowledgements

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] J. Larus and R. Rajwar, “Transactional Memory”, Morgan Claypool, 2006.

[2] OpenMP Architecture Review Board, OpenMP Application Program Interface, May 2005.

[3] T. Harris, M. Plesko, A. Shinnar and D. Tarditi, “Optimizing Memory Transactions”, PLDI ’06: ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, June 2006.

[4] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G.L.Steele Jr. and S. Tobin-Hochstadt. The Fortress Language Specification. Sun Microsystems, 2005.

[5] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun and V. Sarkar. 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), pages 519-538, New York USA, 2005.

[6] Cray. Chapel Specification. February 2005.

[7] N. Shavit and D. Touitou, "Software Transactional Memory", Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, 1995, pp. 204-213.

[8] M. Herlihy and J. Eliot B. Moss, “Transactional Memory: Architectural Support for Lock-Free Data Structures”, In Proc. of the 20th Int’l Symp. on Computer Architecture (ISCA’93), pp. 289-300, May 1993.

[9] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir and D. Nussbaum, "Hybid Transactional Memory", Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 2006.

[10] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, A. Nguyen, “Hybrid Transactional Memory”, In the Proceedings of ACM Symp. on Principles and Practice of Parallel Programming , March 2006.

[11] B. Saha, A. Adl-Tabatabai, Q. Jacobson. “Architectural Support for Software Transactional Memory”, 39th International Symposium on Microarchitecture (MICRO), 2006.

[12] A. Shriraman, V. J. Marathe, S. Dwarkadas, M. L. Scott, D. Eisenstat, C. Heriot, W. N. Scherer III, and M. F. Spear. “Hardware Acceleration of Software Transactional Memory”, TRANSACT 2006.

[13] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé and J. Labarta. “Nanos Mercurium: A Research Compiler for OpenMP”. European Workshop on OpenMP (EWOMP'04). Pp. 103-109. Stockholm, Sweden. October 2004

[14] X. Martorell, E. Ayguadé, N. Navarro, J. Corbalan, M. Gonzalez and J. Labarta, “Thread Fork/join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors”. 13th International Conference on Supercomputing (ICS’99), Rhodes (Greece). June 1999

[15] M. Milovanović, O. S. Unsal, A. Cristal, S. Stipić, F. Zyulkyarov and M. Valero, “Compile time support for using Transactional Memory in C/C++ applications”, 11th Annual Workshop on the Interaction between Compilers and Computer Architecture INTERACT-11, Phoenix, Arizona, February 2007.

[16] A. McDonald, J. Chung, B. Carlstrom, C. Minh, H. Chafi, C. Kozyrakis and K. Olukotun, “Architectural Semantics for Practical Transactional Memory” in *Proc. 33th Annu. international symposium on Computer Architecture, pp. 53-65, 2006.*

[17] W. Baek, C.-C. Minh, M. Trautmann, C. Kozyrakis and K. Olukotun, “The OpenTM Transactional Application Programming Interface”. In *Proc. 16th International Conference on Parallel Architectures and Compilation Techniques (PACT’07)*. Romania, September 2007.

[18] M. Milovanović, R. Ferrer, O. Unsal, A. Cristal, E. Ayguadé, J. Labarta and M. Valero, “Transactional Memory and OpenMP”. In *the Proceedings of the Intl. Workshop on OpenMP*, Beijing/China, June 2007.

[19] N. Sonmez, C. Perfumo, S. Stipic, A. Cristal, O. Unsal and M. Valero, “unreadTVar: Extending Haskell Software transactional Memory for Performance”. In *the Proceedings of the Eight Symposium on Trends in Functional Programming*, New York/USA, April 2007.

[20] Gauss-Seidel Method for solving the linear system of equations. Online material available at: http://en.wikipedia.org/wiki/Gauss-Seidel_method, July 18, 2007.

[21] K. Fraser, Practical lock freedom. PhD thesis, University of Cambridge Computer Laboratory, 2003.