Transactional Access to Shared Memory in StarSs, a Task Based Programming Model

Rahulkumar Gayatri[1,2], Rosa M. Badia[1,3], Eduard Ayguade[1,2], Mikel Luján[4], and Ian Watson[4]

[1] Barcelona Supercomputing Center, Barcelona, Spain
{rgayatri,rosa.m.badia,eduard.ayguade}@bsc.es
[2] Universitat Politècnica de Catalunya, Spain
[3] Artificial Intelligence Research Institute (IIIA), Spanish National Research Council (CSIC), Spain
[4] University of Manchester, UK
{mikel.lujan@manchester.ac.uk,watson@cs.man.ac.uk}

Abstract. With an increase in the number of processors on a single chip, programming environments which facilitate the exploitation of parallelism on multicore architectures have become a necessity. StarSs is a task-based programming model that enables a flexible and high level programming. Although task synchronization in StarSs is based on data flow and dependency analysis, some applications (e.g. *reductions*) require *locks* to access shared data.

Transactional Memory is an alternative to lock-based synchronization for controlling access to shared data. In this paper we explore the idea of integrating a lightweight Software Transactional Memory (STM) library, TinySTM , into an implementation of StarSs (SMPSs). The SMPSs runtime and the compiler have been modified to include and use calls to the STM library. We evaluated this approach on four applications and observe better performance in applications with high lock contention.

1 Introduction

Over the past decade, single-core processors ran into three walls, namely ILP (Instruction Level Parallelism), power and memory. The ensuing stalemate led to the trend of placing multiple slower processors on a single chip. But achieving good performance on these architectures is hard. It often requires programmers to rewrite the code or implement algorithms anew. In multi-core programming, the programmer's efforts are directed towards hardware details, such as movement of data between processors and synchronization, than on the details of the algorithm. Every new architecture additionally comes with its associated SDK, which raises the issue of portability. Hence, what is needed now are programming environments, i.e. sets of compilers, runtimes and communication libraries, that make multi-core programming easier while achieving maximum performance. The effectiveness of such a programming model can be evaluated using the following measures:

- Performance of an application using the programming model versus the native SDK
- Level of complexity exposed to the programmer.
  - Increase in number of lines compared to sequential program
  - Use of specific API calls
- Ease of portability of applications.

OpenMP [5] is a widely used programming model for share memory architectures. It supports multi-platform multiprocessor programming in C, C++, and Fortran. Cilk [7] is a similar programming model developed at MIT. Both OpenMP and Cilk support task-based and loop-based parallelism but neither performs task-based data dependence analysis. Magma [4] is a programming language designed to investigate algebraic, geometric and combinatorial structures. It is not intended for general-purpose programming since its structure is preconditioned for linear algebra problems.

StarSs[9] is a programming environment for parallel architectures such as Symmetric Multiprocessors (SMP), the Cell Broadband Engine (Cell B./E.), Graphical Processing Units (GPU) and clusters. An application written with this programming model, can be executed on any of the architectures mentioned above with no change to the code, effectively achieving portability. In this paper we focus on SMPSs [10], the implementations of StarSs for SMP.

SMPSs allows programmers to write sequential applications, while the runtime exploits the inherent concurrency and schedules tasks to different cores of an SMP. We will have more to say on this topic in Section 2. In order to protect the atomicity of shared memory locations, SMPSs uses *locks*. But locks suffer from the traditional drawbacks of:

- Deadlock - two tasks trying to lock two different objects, each getting access to one and waiting for the other one to be released.
- Livelock - similar to deadlock, except that the state of a livelocked process changes constantly, although without progressing.
- Priority Inversion - a high-priority thread blocked by a low-priority thread.

Software Transactional Memory (STM) is an alternative method to lock-based synchronization for accessing shared-memory locations. To this end a program wraps operations (i.e., reads and writes) in a transaction and STM guarantees that either all the operations in the transaction occur or none. It is a non-blocking approach where a transaction tentatively updates shared memory. If successful it makes the changes permanent and visible to other transactions, else the transaction aborts and restarts [8]. This opportunistic strategy helps us in avoiding problems arising from locks.

There are many TM systems available which allow programmers to access and modify data through transactions. The Intel C++ STM compiler provides extensions to its C++ compiler with support for STM language extensions [1]. RSTM is a set of STM systems available from the Rochester Synchronization Group. It consists of different library implementations and a smart-pointer API for relatively transparent access to STM, and requires no compiler changes. TinySTM

[6] is a word-based STM implementation of the LSA algorithm, available from the University of Neuchatel. In this paper we explore the idea of integrating TinySTM into SMPSs, as a replacement to *locks* for synchronizing simultaneous access to critical memory locations. The rest of the paper is organized as follows: Section 2 explains the basic framework of SMPSs, Section 3 discusses STM and TinySTM, Section 4 presents our idea of integrating TinySTM in SMPSs, Section 5 evaluates and characterizes the performance of our idea. section 6 presents the conclusions and section 7 discusses the future work that we intent to do.

2 SMPSs

SMP Superscalar (SMPSs) consists of a source-to-source compiler and a runtime library. The programmer annotates the sequential code and marks tasks or units of computation using *pragmas* provided by the SMPSs compiler. During execution, the SMPSs runtime analyzes the data accesses of these tasks, but does not immediately perform the corresponding computation. Instead it builds a Task Dependency Graph(TDG), where each node represents a task instance, and edges denote dependencies between tasks. SMPSs uses the TDG to schedule tasks to cores. Independent tasks, i.e. tasks without incoming edges can execute in parallel.

2.1 SMPSs Syntax

As mentioned previously, the programmer typically annotates the functions using pragmas and declares tasks:

```
  #pragma css task [clauses]
function definition / function declaration
```

Listing 1.1. Syntax of a Task Declaration

The clauses indicate the type of access that a task performs for each parameter. Every task parameter must appear in one of the clause, along with its dimensions. SMPSs supports the following clauses in the task pragma:

```
The list of main clauses is the following:
    input ([list of parameters])
    output ([list of parameters])
    inout ([list of parameters])
    reduction ([list of parameters])
```

The runtime builds a TDG based on the directionality of the parameters. Shown below is an example of a task pragma:

```
#pragma css task input(A[NB][NB],B[NB][NB],NB) inout(C[NB][NB])
void matmul(float  *A, float *B, float *C, unsigned long NB)
```

2.2 The *Reduction* Clause

Although the SMPSs runtime only schedules independent tasks for parallel execution, the programming model supports a *reduction*-clause which allows parallel updates to a specified memory regions. The runtime does not insert an edge in the TDG in this case. Responsibility falls on the programmer to access shared memory in the critical section using *lock* and *unlock* pragmas provided by SMPSs, for example:

```
#pragma css task input (n, j, a[n]) inout (results) reduction (results)
void nqueens_ser_task(int n, int j, char *a, int *results)
{
  ....
  #pragma css mutex lock (results)
       *results = *results + local_sols;
  #pragma css mutex unlock (results)
  .....
}
```

Listing 1.2. Example of reduction

In the above Listing 1.2, the reduction applies to the variable *results*, which implies the latter can be updated simultaneously by different tasks. Hence the atomicity of the updates need to be guaranteed by lock and unlock pragmas.

3 Software Transactional Memory

Software Transactional Memory (STM) is an optimistic approach to manage concurrent accesses to shared memory locations. When two different transactions simultaneously try to update the same memory location, STM detects a conflict and allows only one of the transactions to complete successfully. The other transaction is either delayed or aborted. The delaying or aborting of the transaction is also called rollback and the transaction is called the conflicting transaction. The idea was first implemented by Shavit and Touitou [12]. STM simplifies the implementation of shared memory access since each transaction can now be viewed as an isolated series of operations. Every transaction is composed of 4 basic steps:

1. Start of a transaction.
2. Load values from memory into the current transactional environment.
3. Store values back to memory.
4. Commit the results.

Different STM libraries check for conflicts at different steps, depending on their implementation and design. Conflicts can be handled in various ways. The decision of which transaction should be allowed to complete and which should roll back, depends on the contention manager being used.

STM places limitations on the kind of operations that can be executed in a transaction. Only operations that can be rolled back should be employed, whereas for example I/O operations (like *printf("")* in C) cannot be included in a transaction. The use of STM implies a performance degradation due to the overhead incurred in the roll back of transactions.

3.1 TinySTM

TinySTM[6] is a word based STM library based on the *atomic_ops* library[3]. It implements a single version, word-based variant of the Lazy Snapshot Algorithm(LSA)[11]. Like most STM implementations TinySTM uses a shared array of locks to manage concurrent memory accesses. It maps addresses to locks, via a hash function, and uses a shared counter to maintain the timestamp validity of memory locations in transactions. It has three strategies to access memory:

1. WRITE_BACK_ETL - locks are acquired during encounter time.
2. WRITE_BACK_CTL - locks are only acquired during commit.
3. WRITE_THROUGH - directly updates memory and maintains an undo log for rollbacks.

In order to decide which transaction should roll back in case of a conflict, TinySTM has several built in contention managers:

1. CM_SUICIDE - Abort the transaction that detects the conflict.
2. CM_DELAY - Similar to CM_SUICIDE, but the rolled back transaction waits until the contended lock has been released.
3. CM_BACKOFF - Like CM_SUICIDE, but delay restarting the transaction for a random time.
4. CM_AGGRESSIVE - Kill the other transaction.

We use WRITE_BACK_CTL to access memory with TinySTM in SMPSs.

This choice is in line with our future work, where we plan to introduce the speculative execution of tasks in SMPSs. Tasks will execute speculatively, but the committing of the results is postponed to later stages. For handling conflicts we use the CM_DELAY contention manager. CM_DELAY restarts the rolled back transaction when the contended lock is released. As such we avoid the possibility of a transaction being rolled back repeatedly because of the same contention.

4 Integrating TinySTM in SMPSs

In order to incorporate TinySTM library calls in SMPSs applications, the library in question has to be initialized and threads have to be made as transactional threads. The initialization of TinySTM library (*stm_init*) takes place in the main thread of SMPSs and each SMPSs thread registers itself as a transactional thread (*stm_init_thread*). In order to replace *locks* with STM, the operations executed between lock and unlock pragmas must be wrapped in a transaction. The memory locations accessed in this region must be updated using STM calls. An SMP thread finally must commit the result of these operations and make them permanent. For example, the code of Listing 1.2 is transformed into :

```
 1 #pragma css task input (n, j, a[n]) inout (results) reduction (results)
 2 void nqueens_ser_task(int n, int j, char *a, int *results)
 3 {
 4   ....
 5   sigjmp_buf* jump = stm_start(NULL); //start the transaction
 6   if(jump != NULL)
 7     setjmp(*jump, 0); //save stack context
 8   int buffer += stm_load_int(results) + local_sols;
 9   stm_store_int(results, buffer);
10   stm_commit(); //commit transaction
11   ...
12 }
```

Listing 1.3. Implementation of Listing 1.2 with TinySTM library calls

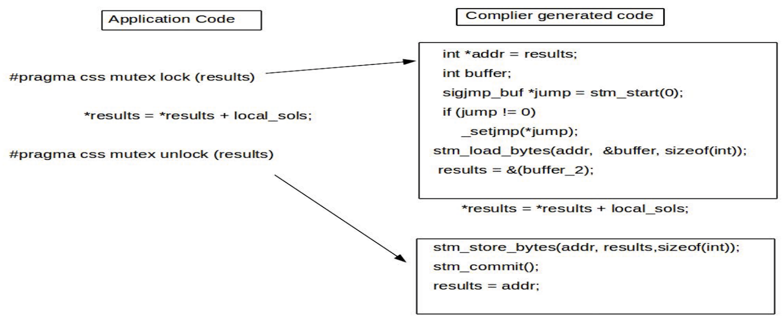
As soon as a transaction starts (line 5 of Listing 1.3), the stack context is saved using a call to *setjmp* (line 7 of Listing 1.3). Critical memory locations are loaded into the current transactional context via calls to TinySTM (line 8 of Listing 1.3). The SMP thread performs the updates and at the end, stores the results and commits the transaction (lines 9 and 10). If it detects a conflict while committing the results, then it performs a *longjmp* and the execution is restarted from the location of the associated *setjmp*. In Listing 1.3., we inserted the transactional calls to the TinySTM library manually. But our objective is to implement transactional access to shared memory locations in SMPSs, without burdening the programmer with these implementation details. TinySTM calls then have to be performed from the SMPSs runtime or inserted by the compiler. Instead of adding new pragmas for this purpose, we decided to modify the implementation of the existing lock and unlock pragmas.

When The SMPSs compiler processes the lock and unlock pragmas, it replaces them with runtime calls to *css_lock* and *css_unlock* respectively. css_lock locks the parameter passed to the lock pragma while css_unlock unlocks it. The basic idea is to start a transaction when a lock pragma is encountered and to commit the results with the corresponding unlock pragma. Its implementation in the SMPSs library is troublesome, due to the use of stack calls of *setjmp* and *longjmp* by TinySTM. This restricts the start and end of a transaction to the same stack frame.

As the locking pragmas map to different functions in the SMPSs runtime, the stack context changes between the occurrence of a lock and its associated unlock. If we modify the runtime such that a transaction is started from css_lock and the results committed in css_unlock, conflicting transaction will not be rolled back correctly.

Instead the SMPSs compiler was modified to insert transactional calls to the TinySTM library. A transaction is started when a thread acquires a lock and the address to be locked is loaded into the transactional environment. The compiler creates a local variable and assigns it the memory address passed to the lock pragma. All updates are performed on this local variable. When the thread releases the lock for the memory location, i.e. at the location of an unlock pragma, the value of the local variable is stored back into the memory address and the transaction is committed. If another thread, and hence a different transaction, has modified the shared memory location since the time it was loaded into the local variable, the transaction is rolled back and restarted.

Below we show how the compiler transforms lock and unlock pragmas for Listing 1.2:



```
#pragma css mutex lock (results)

        *results = *results + local_sols;

#pragma css mutex unlock (results)
```

```
  int *addr = results;
  int buffer;
  sigjmp_buf *jump = stm_start(0);
  if (jump != 0)
     _setjmp(*jump);
stm_load_bytes(addr,  &buffer, sizeof(int));
 results = &(buffer_2);

      *results = *results + local_sols;

stm_store_bytes(addr, results,sizeof(int));
stm_commit();
results = addr;
```

Fig. 1. Compiler generated code of Listing 1.2

The local variable is private to the thread and its scope is the task scope, i.e., this variable is alive only within this instance of the task.

We observed that in some applications it is more efficient to load multiple shared memory addresses into a single transaction rather than to generate a separate transaction for every address. Therefore we extended the *lock* pragma to accept more than one address and load them into a single transactional context. An example of multiple memory locations passed to a single lock pragma is shown in Listing 1.4.

5 Results

To evaluate the performance of our implementation, we executed 4 benchmark programs and the results are compared between SMPSs applications using locks and STM. The applications chosen are:

- NQueens - of placing chess queens on a n*n board such that no queens attack each other. The problem size of the results presented is a chess board of size 14*14 . The problem can have more than one unique solution. The critical section in this application was when updating the number of unique solutions that the problem has.
- Gmeans - a data mining algorithm to find clusters in a dataset. It returns the number of Gaussian distributions and their centers contained in the dataset. The atomicity in this application is required while updating the centers of clusters in the data set.The problem size is a data set of 10 million points of 10 dimensions each. It was observed that the application was not scaling with more than 8 threads.

- Matrix Multiplication - In this application the values of resultant matrix are simultaneously updated by different tasks. Hence, while storing the results a lock needs to be acquired on element of the matrix. The dimensions of the matrix are (128*16) * (128*16).
- Specfem3D - the algorithm simulates seismic wave propagation in sedimentary basins or any other regional geological model. In this application, locks are used in two different stages, once while localizing the data in tasks from global vectors and again while summing the values from each tasks in the global mesh. The global mesh is accessed both directly and indirectly which leads to conflicting accesses of the same position some times.

5.1 Performance Evaluation

The performance evaluation is shown below:

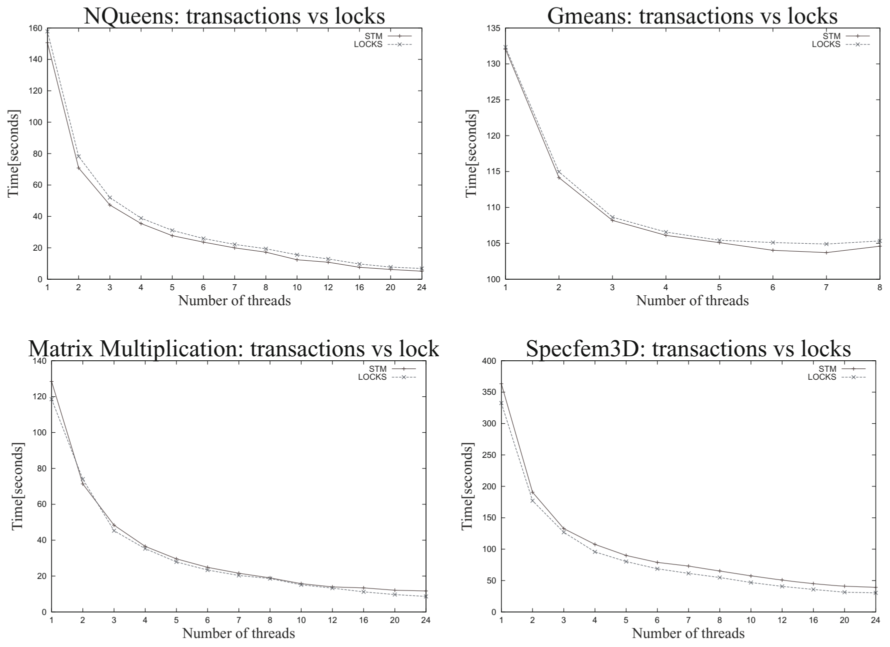


Fig. 2. Performance comparison between SMPSs examples using Transactions and Locks

The above mentioned applications were executed on an Intel(R) Xeon(R) E7450@2.40GHz machine with 24 cores. Thread affinity was controlled by assigning one thread to each core. From Fig.2., we observe that in Nqueens and Gmeans applications STM performs better than locks, whereas in matrix multiplication example, the execution timings are nearly similar. In Specfem3D, even though locks perform better than STM, we observe that the STM version scales. We hope that with further optimizations and better hardware support, STM will have higher performance.

5.2 Performance Characterization

While using STM, with increasing number of threads there is a higher probability of transactions conflicting with each other. Since more threads try to simultaneously update the shared memory locations, more conflicts occur resulting in more rollbacks. Hence we analyzed the behavior of above mentioned applications with increasing number of threads. We used Paraver[2], a flexible visualization tool to analyze characteristics of transactions generated while running the applications. SMPSs runtime generates performance trace-files if *tracing* flag is enabled during compilation. These traces can be analyzed using paraver. Transaction specific events such as time spent in executing operations in a transaction, time spent in commit and rollback were added to paraver tracefile. Shown below is analysis of time spent by applications in different phases of transaction when executed with varying number of threads.

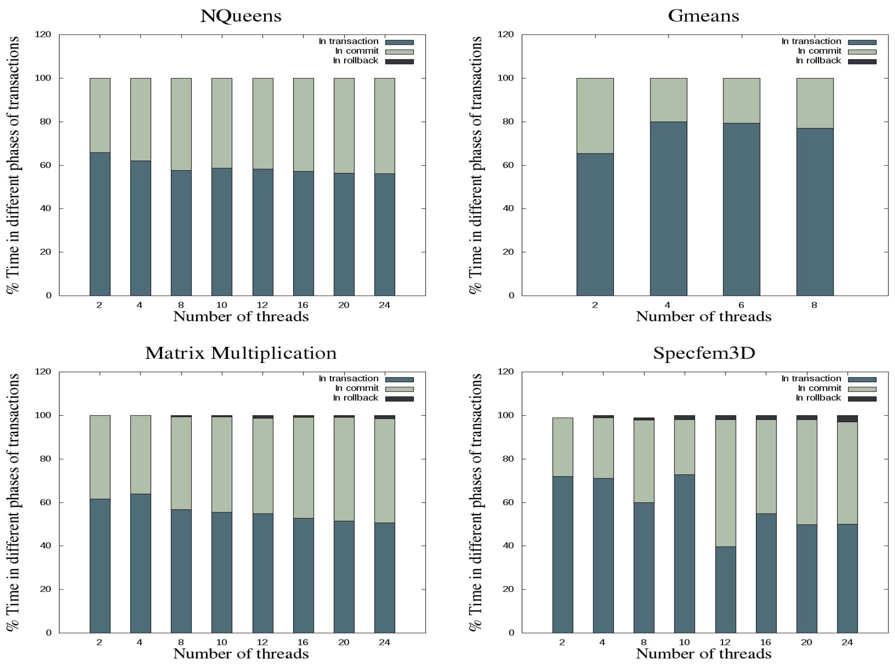


Fig. 3. Time spent in different phases of transaction

From Fig.3., we can observe that in NQueens and Gmeans applications, time spent in rollback is minimal. This is the main reason for their better performance when compared to their lock based implementations. We can also observe that in Gmeans application, threads spend longer time in commit compared to operations executed in transactions. The reason is, since every point in the data set is of multiple dimensions, while committing the results of updated centers the value of each dimension has to be committed. This leads to higher amount of time being spent in commit. We can also observe that matmul and specfem3D spend significantly more time in rollback, compared to the other two benchmarks. As rollback contributes to run-time overhead and thus to the overall execution time, the performance decreases accordingly. However, the rollback overhead does not increase linearly with the number of threads. The potential for contention, and hence rollback, of an application is bounded by the amount of parallelism it admits. The latter is a characteristic of the TDG of an application, not of the number of resources (like the number of threads) with which one chooses to execute. This observation gives us optimism that, with further optimizations and better hardware support, we can further improve the performance of our parallel programming model using STM.

As mentioned earlier sometimes it is more efficient to update multiple shared memory locations in a single transaction compared to generating one transaction for every single address. The trade-off is, to create multiple smaller transactions and thus spend more time in start and commit of transactions versus longer transactions and hence longer time in rollbacks in case of a conflict. In Specfem3D instead of updating 3 different memory locations separately we updated all 3 in a single transaction. Shown below is a snippet of the code.

```
//3 Transactions
#pragma css mutex lock((temp_x))
*(temp_x) += sum_terms[elem][k][j][i][X];
#pragma css mutex unlock((temp_x))

#pragma css mutex lock((temp_y))
*(temp_y) += sum_terms[elem][k][j][i][Y];
#pragma css mutex unlock((temp_y))

#pragma css mutex lock((temp_z))
*(temp_z) += sum_terms[elem][k][j][i][Z];
#pragma css mutex unlock((temp_z))

//1 big transaction
#pragma css mutex lock(temp_x,temp_y,temp_z)
*(temp_x) += sum_terms[elem][k][j][i][X];
*(temp_y) += sum_terms[elem][k][j][i][Y];
*(temp_z) += sum_terms[elem][k][j][i][Z];
#pragma css mutex unlock(temp_x,temp_y,temp_z)
```

Listing 1.4. Specfem3D

We observed that it was more optimal to update three different shared memory locations in a single transaction compared to creating a transaction for each of them. Shown in Fig.4., is how in Specfem3D these two approaches affected the number of rollbacks:

While in this case longer transactions performed better, sometimes they can degrade the performance due to longer rollback time.

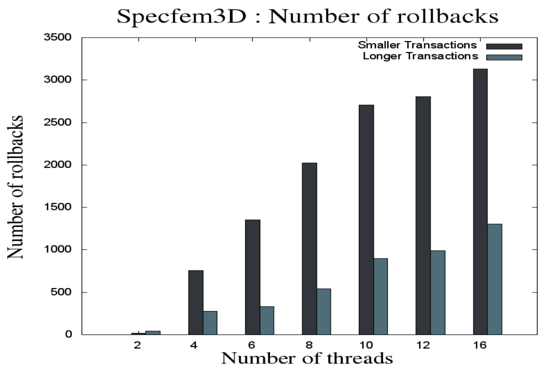


Fig. 4. Specfem3D : Number of rollbacks

6 Conclusion

To keep pace with Moore's law, the trend in the processor industry is to place multiple processors on a single chip. To completely utilize this power, the need of the hour is of programming models offering an easier way to exploit parallelism. StarSs is one such programming model for widely used multicore architectures. It uses *lock* based synchronization during simultaneous updates of shared memory. STM is an alternative shared memory synchronization technique. It is optimistic in nature and simplifies concept of concurrent access to shared memory. We integrate TinySTM, a lightweight STM library with SMPSs (one implementation of StarSs) and replace locks with transactions in SMPSs applications. The results were optimistic with higher performance in applications with high lock contention.

7 Future Work

We plan to use STM to speculatively execute tasks. SMPSs provides programmers with synchronization constructs such as *barriers* and *wait-on*(wait for a particular variable or memory location to be updated before continuing execution). Such constructs lead to problems of load balancing. Hence we plan to introduce speculation using STM. In cases where there is a control dependency between tasks and not data dependency, we can use STM to speculate and execute tasks but postpone the committing of their results till the dependency is resolved.

Acknowledgements. We thankfully acknowledge the support of the European Commission through the TERAFLUX project (FP7-249013) and the HiPEAC-2 Network of Excellence (FP7/ICT 217068), the support of the Spanish Ministry of Education (TIN2007-60625, CSD2007-00050 and FI program) and the Generalitat de Catalunya (2009- SGR-980).

References

1. http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition-20/
2. http://www.bsc.es/media/1364.pdf
3. http://www.hpl.hp.com/research/linux/atomicops/
4. magma.maths.usyd.edu.au/magma/pdf/intro.pdf
5. Duran, A., Perez, J.M., Ayguadé, E., Badia, R.M., Labarta, J.: Extending the OpenMP Tasking Model to Allow Dependent Tasks. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 111–122. Springer, Heidelberg (2008)
6. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: PPoPP 2008, Salt Lake City, Utah, USA. ACM (2008)
7. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the cilk-5 multithreaded language. In: Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 1998, Montreal, Canada. ACM (1998)
8. Harris, T., Larus, J., Rajwar, R.: Transactional Memory, 2nd edn. Morgan and Claypool Publishers (2010)
9. Perez, J.M., Badia, R.M., Labarta, J.: A dependency-aware task-based programming environment for multi-core architectures. In: IEEE Int. Conference on Cluster Computing, pp. 142–151 (September 2008)
10. Perez, J.M., Badia, R.M., Labarta, J.: Handling task dependencies under strided and aliased references. In: Proceedings of the 24th ACM International Conference on Supercomputing, ICS 2010, pp. 263–274. ACM, New York (2010)
11. Riegel, T., Felber, P., Fetzer, C.: A lazy snapshot algorithm with eager validation. In: Proceedings of the 20th International Symposium on Distributed Computing
12. Shavit, N., Touitou, D.: Software transactional memory. In: 14th ACM Symposium on the Principles of Distributed Computing, Ottowa, Ontario, Canada. ACM (1995)