Analysis of the overheads incurred due to speculation in a task based programming model.

Rahulkumar Gayatri[§ †], Rosa M. Badia[§ ♭], Eduard Ayguade [§ †]

[§] Barcelona Supercomputing Center [†] Universitat Politècnica de Catalunya,
[♭]Artificial Intelligence Research Institute (IIIA), Spanish National Research Council (CSIC), Spain
{rgayatri, rosa.m.badia, eduard.ayguade}@bsc.es

Abstract. In order to efficiently utilize the ever increasing processing power of multi-cores, a programmer must extract as much parallelism as possible from a given application. However with every such attempt there is an associated overhead of its implementation. A parallelization technique is beneficial only if its respective overhead is less than the performance gains realized. In this paper we analyze the overhead of one such endeavor where, in SMPSs, speculation is used to execute tasks ahead in time. Speculation is used to overcome the synchronization pragmas in SMPSs which block the generation of work and lead to the underutilization of the available resources. TinySTM, a Software Transactional Memory library is used to maintain correctness in case of mis-speculation. In this paper, we analyze the affect of TinySTM on a set of SMPSs applications which employ speculation to improve the performance. We show that for the chosen set of benchmarks, no performance gains are achieved if the application spends more than 1% of its execution time in TinySTM.

1 Introduction

Currently eight-core chips are highly prevalent in the market and this trend is on a rise. To effectively use such hardware processing power an application should be as parallel as possible. But parallel programming is laborious due to the lack of standardization and portability issues. SMPSs [1] is a task-based programming model which allows a programmer to write high-level parallel code for Symmetric Multiprocessors (SMPs). It consists of a source-to-source compiler and a runtime that comprises of a main-thread and a pool of worker-threads. The compiler processes the directives which are used to annotate parts of a sequential code as individual units of computation (tasks). The main-thread builds a task dependency graph (TDG) based on the data-flow between the tasks and independent nodes from this graph are scheduled to the worker-threads. The separation of the generation and the execution of a task makes it necessary to use the synchronization directives such as barrier and wait-on to maintain control between the main-thread and the worker-threads. The use of such directives, however block the main-thread until the synchronization-predicate has been evaluated

which leads to underutilization of the available resources. Such inabilities to effectively use the available resources become more prominent as the number of cores on a chip keep increasing. To overcome this problem, the SMPSs framework provides a `speculate` directive which avoids the synchronization pragma and generates tasks speculatively. The speculatively generated (speculative) tasks posses a transaction-like property, where their updates are committed or undone based on the success of the speculation.

To achieve the commit or abort type of behavior for the speculative tasks, a lightweight STM [5] library, TinySTM [4] is integrated into the SMPSs framework. SMPSs executes a *speculative* task as a transaction and commits its results only if the speculation has been successful, else the transaction is aborted. Speculation increases the parallelism that can be extracted from a given application. But its impact on the overall execution time of an application is also dependent on the overhead associated with its implementation. We measure the time spent in the TinySTM library to evaluate the overhead incurred due to speculation in SMPSs. For this we use Paraver [3], a performance analysis tool. We add paraver specific events to measure the time spent in the TinySTM library calls. We compare the execution times of the speculative and non-speculative versions of a set of benchmarks chosen from the domain of linear algebra and graph algorithms. We analyze the overhead of executing TinySTM library calls in the applications and their effects on the overall performance. We also present the behavior of the speculative versions of the benchmarks in the presence of weak and strong scaling. The analysis provides us with an upper-bound for the acceptable overhead with speculation. The contributions of this paper are:

- An empirical proof to show that the benefits of speculation improve with higher number of threads.
- An experimental demonstration that shows the negative effects of unoptimized number of calls to the TinySTM library.
- The effect of invalid tasks on the performance of speculation.

The rest of the paper is organized as follows: in section 2 we discuss the SMPSs framework and the need for speculation in SMPSs and its implementation. We also explain the *speculate* pragma and its usage. In section 3 we present our evaluation on the various aspects that effect the efficiency of speculation. In section 5 we present our conclusions.

2 Speculation in SMPSs

The SMPSs framework provides compiler directives (pragmas) to annotate code blocks in a sequential program that can potentially be run in parallel. The annotated code blocks are treated as units of computation, tasks. The pragmas are processed by the SMPSs source-to-source compiler and the transformed code is passed to the SMPSs runtime. The SMPSs main thread builds a task-dependency graph (TDG) to honor the data-dependencies between the tasks. Independent

tasks from this graph are scheduled to the various execution resources. The separation of task generation and its execution makes it impossible for the main-thread to make any assumptions about its completion. Hence in-order to maintain control between the main-thread the framework provides synchronization pragmas such as:

```
#pragma css wait on(a)
#pragma css barrier
```

Listing 1.1: Synchronization pragmas in SMPSs

The first pragma in listing 1.1 halts the main thread until the last task that updates **a** has finished execution whereas the second one halts the main thread until all previously generated tasks have finished execution.

Synchronization pragmas limit the parallelism of a given application as they block the main thread from task generation. In a while-loop, if the loop-predicate is updated in a task inside the loop, a synchronization pragma is used at the end of the loop iteration.

```
while(residual > threshold)
{
  for(int i=0 ; i < x_blocks-1 ; i++)
      for(int j = 0 ; j < y_blocks ; j++)
    jacobi_task(i,j,parameters[i*np+j*block_y],residual);
   #pragma css wait on (residual)
}
```

Listing 1.2: while loop in the Jacobi application.

Listing 1.2 shows an example where a wait pragma in a while-loop does not allow the generation of tasks from more than one iteration at a time. The synchronization is necessary since the tasks that evaluate the predicates could potentially be executed on different threads. To overcome this drawback, a speculate directive [2] has been introduced to the SMPSs framework and is used as follows:

```
#pragma css speculate values(a,b) wait(x)
```

Listing 1.3: Speculate Pragma

- **values** - tasks that update parameters **a,b** should be protected in case the speculation fails.
- **wait** - indicates the predicate that decides the commit or abort of the speculative tasks.

If the *speculate* pragma is inserted before a programming construct, then the tasks inside these constructs will be generated speculatively. In this context we define "invalid tasks" as tasks where the speculation fails and hence their updates should be undone, i.e., such tasks should be rolled back. Currently the speculate pragma can be used with the following programming constructs : while-loop and if-condition. Figure 1 shows the usage of the speculate pragma. The speculative

if condition

```
.....
Task1(x,y);
#pragma css speculate \
values (a,b) wait(x)
#pragma css wait on(x)
if(x)
{
  for(;;)
  {
    Task2(a,b);
    Task3(c,d);
  }
}
...
```

while loop

```
.....
Task1(x,y);
#pragma css speculate \
values (a,b) wait(x)
while(x)
{
  for(;;)
  {
    Task2(a,b);
    Task3(c,d);
  }
#pragma css wait on(x)
}
....
```

Fig. 1: Usage of the **speculate** pragma.

tasks apart from being control dependent on the predicate of the loop may also be data dependent on the earlier tasks. Hence the use of the *speculate* pragma may affect the TDG of the application in one of the following ways:
1) Tasks which are control-dependent on the earlier tasks.
2) Tasks which are data-dependent on the earlier tasks.

In case 1, simultaneous execution of speculative tasks with earlier (speculative and non-speculative) tasks is possible. In case 2, the SMPSs main thread adds dependencies between speculative tasks and earlier tasks. This only allows the overlap of speculative task generation with task execution. We evaluate and analyze the performance of the applications and the behavior of the SMPSs and TinySTM runtimes when the *speculate* pragma is used.

3 Performance Analysis.

The performance analysis was done on 4 different applications namely:
1) Lee-routing - generates control-dependent tasks when speculate directive is used.
2) Gauss-Seidel and Jacobi - iterative solvers which generate *speculative* tasks which are data-dependent on the earlier tasks.
3) Kmeans - data-clustering algorithm which also generates data-dependent *speculative* tasks.

The above mentioned applications are executed on an IBM dx360 M4 node. It contains 2x E5-2670 SandyBridge-EP 2.6GHz cache 20MB 8-cores. Thread affinity was controlled by assigning one thread to each core. The applications which generate data-dependent *speculative* tasks were executed with three different problem sizes which are shown in table 1.

Problem-sizes	Gauss-Seidel	Kmeans	Jacobi
Small	2048 unknowns	100 thousand	2048 unknowns
Medium	4096 unknowns	500 thousand	4096 unknowns
Large	8192 unknowns	1 million	8192 unknowns

Table 1: Problem sizes of the benchmarks chosen.

We concentrate more on applications where the *speculate* pragma generates tasks that are data-dependent on earlier tasks, since they occur with more frequency in programming. The use of an external library effects the performance of SMPSs since it now requires to execute function calls outside of its framework. Hence we focus our investigation of STM-based speculation on the following points:

- We compare the performances of speculative and non-speculative versions of the applications.
- Effect of varying the task granularities on the performance of applications.
- Relative time spent in the TinySTM library compared to the total execution time.
- Overhead of invalid tasks.

We study the case of lee-routing application separately since this is the only application that adds *speculative* tasks which are independent of the tasks generated earlier. Figure 2 shows the performance comparison of the phase of Lee-routing where the tasks are blocked due to a synchronization pragma that enforces a control dependency.

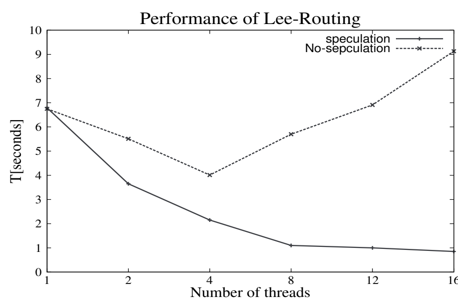


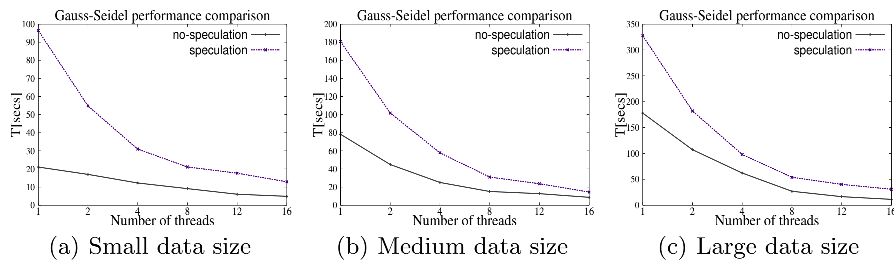
Fig. 2: Lee Routing.

The parallelism extracted from the simultaneous execution of the speculative tasks and the tasks generated earlier is evident from the gain in the performance. In this phase, the non-speculative version does not improve the performance after 4 threads. Instead the performance worsens due to the extra resources that are unnecessarily used and the overhead of the SMPSs framework wasted

when there is no parallelism. The speculative version, however steadily improves in the performance timings until 8 threads. Even after that when scaled, the performance does not degrade but remains similar without much improvement either.

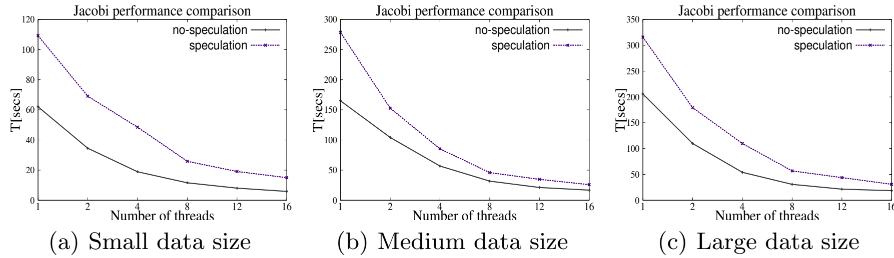
Performance Comparisons

Figures 3, 4 and 5 show that the speculative version of the Gauss-Seidel, Jacobi and Kmeans never perform better compared to their non-speculative counterparts. Irrespective of the number of threads used and the problem size, the reg-



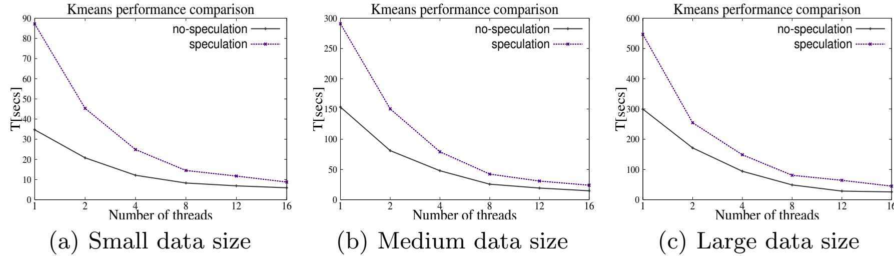
(a) Small data size (b) Medium data size (c) Large data size

Fig. 3: Performance comparisons of Gauss-Seidel for all three problem sizes.



(a) Small data size (b) Medium data size (c) Large data size

Fig. 4: Performance comparisons of Jacobi for all three problem sizes.



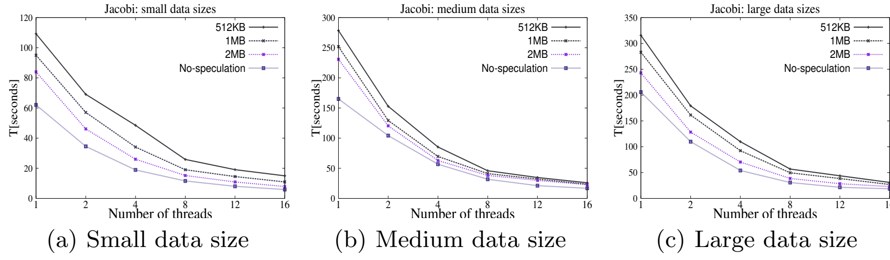
(a) Small data size (b) Medium data size (c) Large data size

Fig. 5: Performance comparisons of Kmeans for all three problem sizes.

ular SMPSs versions consistently performs better. The TinySTM library calls

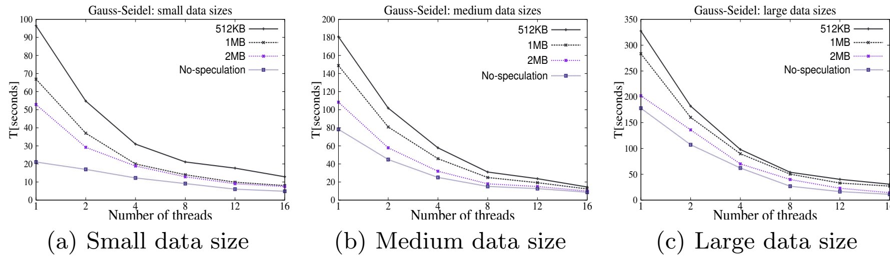
that are used to maintain correctness is an overhead absent in the regular SMPSs application. The execution of transaction calls within every *speculative* task degrades the overall performance of the application. Figures 3, 4 and 5 show that although the speculative version scales similar to its non-speculative counterpart, it never benefits in the absolute performance. The increase in the number of tasks generated does not provide enough boost to the execution timings of the applications in-order to overshadow the TinySTM overheads. However, with an increase in the number of threads the absolute difference in the performances between the speculative and non-speculative version reduces. With higher number of threads, more resources are available to avail the parallelism extracted from the `speculate` clause. This is a positive result, since it proves that the idea is scalable. And if the overhead is reduced, performance gains can be achieved by the idea of speculation.

Varying task granularities
The task granularity, i.e., the size of the memory block that is updated by a single task is chosen as 512KB of data. The choice of task granularity plays an important factor in the performance of an application. Smaller task granularities will lead to an increase in the overhead of task creation and destruction, whereas larger tasks will not be able to make the best use of the concurrency available in the application. The results shown in the previous section used the task granularities that gave the best performance with the non-speculative version of the applications. We now show the effect of modifying task granularities on the application performances. The effects of higher task granularities on the TDG of an application and correspondingly on the calls made to the TinySTM library when the *speculate* clause is used are:
1) Decrease in the number of *speculative* tasks and consequently the number of transactions generated for a given problem size.
2) Increase in the size of the transactional copy performed by a single *speculative* task.
Figures 6, 7 and 8 show the changes in the performances of the Jacobi, Gauss-Seidel and Kmeans with different task granularities. The legend in the figures shows the task granularity. The performance of the non-speculative versions of the application with their optimal task granularities is shown for comparison. In all the three applications for all the three problem sizes, the best performance of the speculative version is achieved with the task-granularity of 2MB. This is the largest among the granularities chosen. Bigger task-granularities decrease the number of tasks generated but increase the amount of data processed within a single task. In case of *speculative* tasks, this leads to a decrease in the number of transactions created and destroyed but an increase in the data processed within a single load/store operation. This leads us to the conclusion that, with TinySTM the overhead of generating more number of smaller transactions is higher than creating less number of bigger transactions. The conclusion is valid since we compare the performance with the optimal task granularity of the non-speculative version of the applications. Although with larger transactions more time is spent



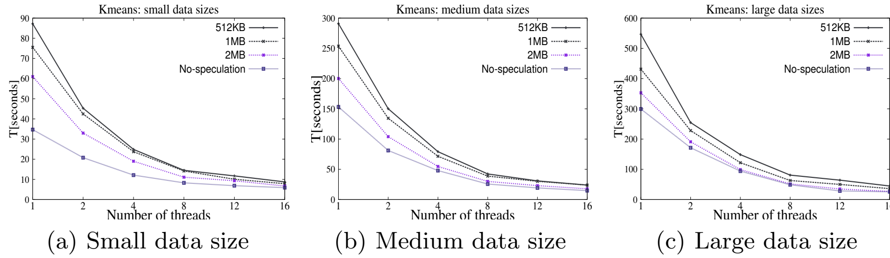
(a) Small data size (b) Medium data size (c) Large data size

Fig. 6: Performance of Jacobi with different task granularities.



(a) Small data size (b) Medium data size (c) Large data size

Fig. 7: Performance of Gauss-Seidel with different task granularities.



(a) Small data size (b) Medium data size (c) Large data size

Fig. 8: Performance of Kmeans with different task granularities.

in rollbacks when the speculation fails, the granularities of the *speculative* tasks should be higher than the regular SMPSs tasks. This is in direct conflict with the purpose of speculation, i.e., to increase the parallelism from a given application, which in the case of SMPSs translates to increasing the number of tasks in the TDG for the worker-threads to choose from.

Time spent in TinySTM

The use of TinySTM to maintain correctness of a *speculative* task, hampers the overall performance of the application. Hence to inspect the effect of TinySTM, we analyze the time spent in different phases of a transaction during the execution of a *speculative* task. In this section, we present the relative time spent by the benchmarks in TinySTM while executing their speculative versions. Our

aim is to obtain an upper bound for the acceptable overhead associated with TinySTM based implementation of speculation. The minimum performance that can be achieved in speculation is the overlap of task generation with task execution. Hence finding an acceptable overhead for applications with cross-iteration dependencies can be safely generalized to all cases.

To calculate the amount of time spent in the TinySTM library during the execution of a speculative task, we trace the execution of the following TinySTM library calls:

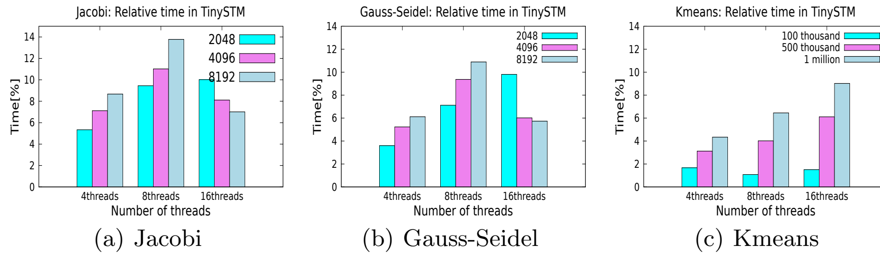
- Start a transaction (stm_start).
- Load/store memory blocks into the transactional context (stm_load_bytes/ stm_store_bytes).
- Abort the transaction in case of mis-speculation (stm_abort).
- Commit the transaction (stm_commit).

We use paraver [3] to perform this analysis and visually analyze the characteristics of the TinySTM library in SMPSs. Trace-events have been added to the SMPSs framework, which track the execution of the above mentioned TinySTM library calls. These events contain the information that represent the cumulative amount of time spent in the respective library calls. Paraver later analyzes these events to deduce the information in a readable visual format.

Figure 9 presents the relative time spent in the TinySTM library while executing *speculative* versions of the applications. The y-axis shows the relative time spent in TinySTM compared to the total execution time of the application. The legend in the figure represents the problem sizes of the application. The task-granularity chosen gives the best performance results for the speculative versions of the applications. An increase in the number of resources increases the number of *speculative* tasks scheduled in parallel (which also include invalid tasks). This will decrease the time taken to execute an application which will consequently increase the relative time spent in TinySTM. Hence longer histograms can mean one or both of the following:

- An increase in the number of invalid tasks due to increase in the number of threads and/or problem sizes of the application.
- Faster execution of the applications, which increases the relative time spent in TinySTM.

All three applications spend different amounts of time in the TinySTM library. The pattern of relative time is similar in Jacobi and Gauss-Seidel, but the amount of time is different. The difference in the iterative solvers allows Gauss-Seidel to converge faster but generates more tasks in every iteration of Jacobi. This implies that the time spent in TinySTM by Gauss-Seidel is influenced by its faster convergence whereas in the case of Jacobi, higher number of invalid tasks is more dominant. Since Jacobi spends more time in TinySTM, it implies that the presence of invalid tasks has more impact on the performance of the speculative versions of the applications. In the results presented the least overhead occurs when the Kmeans application is executed with smaller data size. Even here the



(a) Jacobi (b) Gauss-Seidel (c) Kmeans

Fig. 9: Relative time spent in the TinySTM library.

performance of speculative version is less than the regular SMPSs version. This shows that no performance benefits are gained when the overhead of TinySTM is more than 1% of the total execution time of the application.

An STM library tracks the memory locations that are accessed and updated inside a transaction. If a parallel transaction updates the same memory location then a conflict is detected and one of the transactions is aborted. The data analysis performed by the TinySTM library is unnecessary in the case of SMPSs due to the presence of a TDG for every application. This makes the use of TinySTM or any other STM library to implement speculation unnecessary since its major feature will be redundant. If a regular data-version based implementation is used instead of an STM based one, we obtain performance benefits from the idea of speculation as shown in figure 10. A detailed description about this implementation is presented in [10].

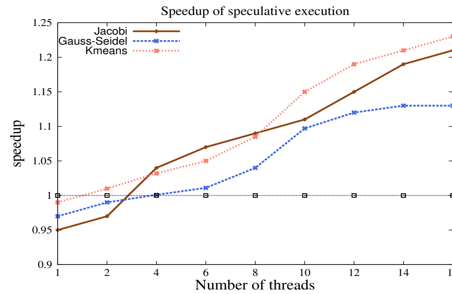
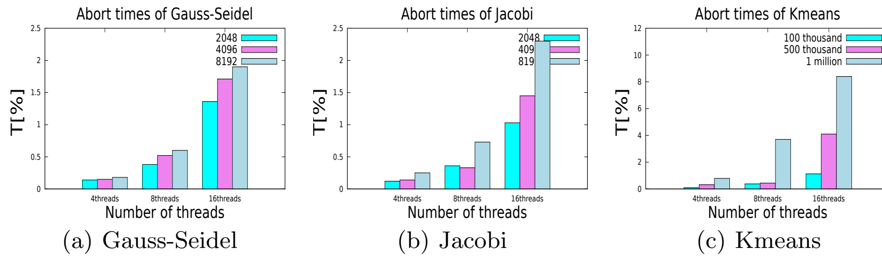


Fig. 10: data-version based speculation implementation.

Invalid tasks

One of the major overheads in speculative task execution is the rollback of invalid tasks. Figure 11 shows the relative time spent by applications in rollbacks, which is shown in the y-axis of the figure. In the figure we observe that with increasing number of threads the relative time spent in aborts increases. Increasing the



(a) Gauss-Seidel (b) Jacobi (c) Kmeans

Fig. 11: Abort times relative to the execution time of the applications.

number of threads is advantageous for speculative execution since it provides more resources to benefit from the increased parallelism. This implies that the applications will execute faster which will increase the relative time spent in rollback. Figure 11 also shows an increase in the problem size also increases the relative time in abort. Increase in the number of tasks generated in every iteration consequently increases the number of invalid tasks in case of mis-speculation. Hence the time spent in rollbacks increase with increasing problem size and the number of threads. The differences in the Gauss-Seidel and Jacobi algorithms can be observed in the abort times too. Jacobi spends more time in aborting invalid tasks compared to Gauss-Seidel. Compared to Jacobi and Gauss-Seidel, the relative time spent in aborting invalid tasks is higher with increasing threads in Kmeans. A comparison with the performance of Kmeans shows that it has least amount of absolute difference with its non-speculative counterpart. It faster execution has increased the relative time spent in aborting invalid tasks.

4 Related Work

Task-wait is a synchronization directive used from OpenMP 3.1 [11] which is similar to `wait` of the SMPSs. It blocks the progress of the current task until all its children tasks have completed its execution. The idea of speculative loop execution is a widely researched topic, where different techniques have been proposed like [13], [14], [15]. But we believe that ours is the first work which implements and analyzes speculative task execution in a programming model and uses STM to maintain correctness.

5 Conclusion

To overcome the problems arising from the use of synchronization directives, SMPSs-framework provides a `speculate` pragma. When used, this pragma generates and executes tasks ahead in time. To maintain correctness, *speculative* tasks are executed as transactions using the TinySTM library calls. To gain performance benefits from this idea the overhead incurred due to TinySTM should be

less than the performance gains achieved by speculation. In this paper we compare the performances of speculative and non-speculative versions of a chosen set of benchmarks and the effect of TinySTM in SMPSs. We conclude that in loops with loop-carried dependencies, no performance benefits can be achieved if the overhead of TinySTM is more than 1% of the total execution time.

References

1. Josep M. Perez Rosa M. Badia Jesus Labarta "Handling task dependencies under strided and aliased references" Proceeding ICS '10 Proceedings of the 24th ACM International Conference on Supercomputing
2. Rahulkumar Gayatri, Rosa M. Badia Eduard Ayguade "Loop Level Speculation in a Task Based Programming Model" Proceeding HiPC'03 Proceedings of the 20th IEEE International Conference on High Performance Computing.
3. "Parallel Program Visualization and Analysis Tool" http://www.bsc.es/media/1364.pdf
4. Pascal Felber, Christof Fetzer, Torvald Riegel "Dynamic Performance Tuning of Word-Based Software Transactional Memory" PPoPP'08
5. Shavit, Nir, and Dan Touitou. "Software transactional memory." Distributed Computing 10.2 (1997): 99-116.
6. Augonnet, Cédric, et al. "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures." Concurrency and Computation: Practice and Experience 23.2 (2011): 187-198.
7. Blumofe, Robert D., et al. "Cilk: An efficient multithreaded runtime system." Vol. 30. No. 8. ACM, 1995.
8. Gottschlich, Justin E. and Connors, Daniel A. "DracoSTM: A Practical C++ Approach to Software Transactional Memory" Proceedings of the 2007 Symposium on Library-Centric Software Design
9. Dragojević, Aleksandar and Guerraoui, Rachid and Kapalka, Michal "Stretching Transactional Memory"
10. Gayatri, Rahul K. and Badia, Rosa M. and Ayguade, Eduard "Loop Level Speculation in a Task-Based Programming Model". Proceedings of the 2013 IEEE conference on High Performance Computing.
11. Dagum, Leonardo, and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming." Computational Science & Engineering, IEEE 5.1 (1998): 46-55.
12. `magma.maths.usyd.edu.au/magma/pdf/intro.pdf`
13. Tian, Chen, et al. "Copy or discard execution model for speculative parallelization on multicores." Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2008.
14. Ding, Chen, et al. "Software behavior oriented parallelization." ACM SIGPLAN Notices. Vol. 42. No. 6. ACM, 2007.
15. M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. " Optimistic parallelism requires abstractions."In ACM SIGPLAN Notices, ACM, 2007.