Loop Level Speculation in a Task Based Programming Model

Rahulkumar Gayatri
Barcelona Supercomputing Center
Barcelona, Spain
Email: rgayatri@bsc.es

Rosa. M Badia
Barcelona Supercomputing Center
Barcelona
Artificial Intelligence Research Institute (IIIA),
Spanish National Research Council (CSIC), Spain
Email: rosa.m.badia@bsc.es

Eduard Aygaude
Barcelona Supercomputing Center
Barcelona, Spain
Universitat Politècnica de Catalunya,
Spain
Email: eduard.aygaude@bsc.es

***Abstract*—Uncountable loops (such as while loops in C) and if-conditions are some of the most common constructs in programming. While-loops are widely used to determine the convergence in linear algebra algorithms or goal finding problems from graph algorithms, to name a few. In general while-loops are used whenever the loop iteration space, the number of iterations a loop executes is unknown. Usually in while-loops, the execution of the next iteration is decided inside the current loop iteration (i.e. the execution of iteration *i* depends on the values computed in iteration *i-1*). This precludes their parallel execution in today's ubiquitous multi-core architectures. In this paper a technique to speculatively create parallel tasks from the next iterations before the current one completes is proposed. If consecutive loop-iterations are only control dependent, then multiple iterations can be executed simultaneously; later in the execution path, the runtime system will decide to either commit the results of such speculatively executed iterations or undo the changes made by them. Data dependences within or between non-speculative and speculative work are honored to guarantee correctness. The proposed technique is implemented in SMPSs, a task-based dataflow programming model for shared-memory multiprocessor architectures. The approach is evaluated on a set of applications from graph algorithms and linear algebra. Results are promising with an average increase in the speedup of 1.2x with 16 threads when compared to non speculative execution of the applications. The increase in the speedup is significant, since the performance gain is achieved over an already parallelized version of the benchmarks.**

***Keywords*-SMPSs; Iteration space; Speculation; Programming Models;**

I. INTRODUCTION

Loops and conditionals are some of the most commonly used programming constructs, of which *while-loops* and *if-conditions* are predominant. As any loop can be converted into a while-loop, our current work is focused on *while-loops* and *if-conditions*. Goal finding problems and convergence algorithms like the Jacobi method and Gauss-Seidel [2] from linear algebra are some of the most common areas where while-loops are used. Such loops run until either a goal is found or some threshold is reached. This implies that the number of iterations that the loop executes is unknown until termination. An attempt to parallelize such loops, without the knowledge of the loop-iteration space, leads to the sequential execution of these iterations. The main reason for the sequential execution of the loops is the inability to predict the execution of iteration *i* until some values from iteration *i-1* have been updated. Hence even though in practice these loops execute for multiple iterations before termination, they cannot be executed in parallel. For example, consider the following pseudo-code of a while-loop which executes until a certain goal is found:

```
while(!goal_achieved)
{
   newBFD = pop_queue();
   subst(refCFG, newBFD, newCFG);
   dimemas(newCFG, trace, dimOUT);
   extract(newBWD, dimOUT, finalOUT);
   check(finalOUT, goal_achieved);
}
```

Listing 1: Example pseudo code

Every iteration of the while loop in Listing 1, pops a new element *newBFD* from a queue. The **subst** function uses *newBFD* and a configuration parameter, *refCFG* as input to generate a new configuration parameter called *newCFG*. The **dimemas** function uses *newCFG* and a trace parameter to generate an output called *dimOUT*. The **extract** function then uses *newBFD* and *dimOUT* to generate the final output called *finalOUT*. A **check** is then performed to evaluate whether *finalOUT* is the goal that is being searched. The loop terminates when the goal is reached.
The functions in a single iteration of Listing 1 are dependent on each other and hence should be executed sequentially to maintain correctness. But since every iteration is based on an element popped inside the iteration, consecutive iterations are independent of each other. This implies that multiple iterations can be executed in parallel. In effect the while-loop shown in Listing 1 consists of only inter-loop parallelism but not intra-loop parallelism. In most cases, such a loop runs for multiple iterations before termination, but no information can be obtained about the available parallelism due to the lack of knowledge of the loop-iteration space. On multi-core architectures such sequential execution of loop-iterations hamper parallelism and lead to an under-utilization of the available resources. An ideal case would be to execute multiple iterations of the loop simultaneously.

But when the loop-predicate evaluates to false, i.e., when the goal_achieved evaluates to true, the information should be propagated across the iterations and the loop should be terminated.

To overcome this problem we implement and evaluate the technique of speculative generation and execution of loop iterations ahead of time. If there are loop carried dependencies between iterations of the loop, i.e, if values produced in iteration i-1 are consumed in iteration i, then the execution of iteration i-1 can only be overlapped with generation of iteration i. In the cases where consecutive loop iterations are independent, as in Listing 1 iteration i and iteration i-1 can be simultaneously executed. But the results from iteration i should be committed only after its validity is confirmed. The idea is an extension to our previous work on speculative updates to shared memory locations using Software Transactional Memory (STM) [7].

We implement our idea in StarSs[10], a task based programming model with support for heterogeneity. StarSs has implementations for widely used multi-core architectures such as Symmetric Multiprocessors (SMP), the Cell Broadband Engine (Cell B./E.), Graphical Processing Units (GPU) and clusters. In this paper we focus on SMPSs, an implementation of StarSs for Symmetric Multiprocessors. Programmers write sequential applications and annotate parts of the code as units of computation or tasks. The SMPSs runtime exploits the inherent parallelism. The SMPSs framework is explained in more detail in Section 2. To evaluate a valid loop predicate for the next iteration of the loop, a synchronization is used at the end of the iteration. The use of the synchronization pragma blocks the task generation at the end of the loop iteration. This inability to generate tasks from multiple iterations restricts parallelism. Hence we speculatively generate tasks from iterations ahead in time. In the rest of the paper we use the terminology, *speculative tasks* for tasks that are speculatively generated. Depending on the type of loop-carried dependencies, *speculative tasks* are either simultaneously executed or blocked until their dependencies have been resolved. If the loop iterations are control dependent, then tasks from consecutive iterations are executed simultaneously, but committing the results of such speculative tasks is postponed until later stages of execution when their validity is confirmed.

The main contributions of the paper are :

- Speculative generation and execution of tasks from multiple loop iterations.
- Overlapping task generation with task execution in case of synchronization pragmas. This reduces the overhead of task generation which is not present in the sequential execution.
- Evaluation of the performance on graph algorithms and linear algebra applications.

The work presented in this paper is organized as follows: Section 2 explains the SMPSs programming model. Section 3 discusses in detail the need for speculation in SMPSs. Section 4 discusses at a higher level, the changes made to the SMPSs compiler and the runtime to introduce speculative generation and execution of tasks. Section 5 presents the results and detailed analysis of applications from the domain of graph algorithms and linear algebra. We evaluate our idea of speculative task generation on these applications. Section 6 presents our conclusions.

II. SMPSs

SMP Superscalar (SMPSs)[11] is a task based programming model. It is based on data flow analysis done at the runtime. SMPSs consists of a source-to-source compiler and a runtime. The programmer writes sequential code and annotates parts of the code that can potentially be executed in parallel. SMPSs compiler provides pragmas for such annotations. These annotated parts of code are treated as tasks or independent units of computation. The annotation of tasks does not guarantee their parallel execution. The SMPSs runtime analyzes the data dependencies between the tasks and schedules them only after their dependencies have been resolved.

A. SMPSs Syntax

The syntax for annotating tasks is:

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

Listing 2: Syntax of a Task Declaration

The clauses indicate the directionality of the parameters passed to the task. The types of clauses supported by the SMPSs compiler are:

```
input ([list of parameters]) //read only
output ([list of parameters]) //write only
inout ([list of parameters]) // read and
    write
reduction ([list of parameters]) // allows
    parallel updates
```

Listing 3: Clauses in a task declaration

In Listing 4, *subst* function from Listing 1 is annotated as a task

```
#pragma css task input(refCFG,newBFD) \
       output(newCFG)
void subst(refCFG, newBFD, newCFG);
```

Listing 4: Example of a task pragma

Since refCFG and newBFD are used to produce newCFG in the *subst* task, they are passed to input and output clauses respectively.

Reduction clause in SMPSs relaxes the dependency tracking for memory locations passed in this clause. This allows the tasks to concurrently update those memory locations.

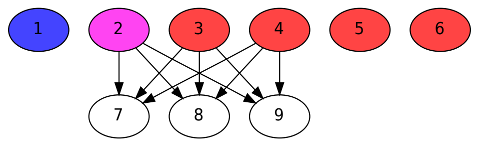

Figure 1: TDG of Kmeans.

Protection of such parallel updates is the responsibility of the programmer. In our previous work [7], we explored the idea of speculatively updating shared memory locations in SMPSs. Technique of optimistic updates to the shared memory using Software Transactional Memory (STM)[9] versus pessimistic updates using locks was evaluated and analyzed. The results showed better performance of STM when used in applications with high lock contention. The current work is an extension from the *speculative* updates performed on shared memory to the *speculative* generation and execution of tasks.

B. SMPSs Runtime

SMPSs runtime comprises of a main thread and multiple worker threads. The main thread of SMPSs executes the application code and builds a Task Dependency Graph (TDG) based on the data accesses performed by the tasks. The data flow analysis used to build the TDG is done based on the directionality information of the parameters passed to the task through its clauses. The TDG consists of nodes which represent a single instance of a given task and edges, whose directions denote the data dependencies between the tasks. An edge from task A to task B indicates a data dependency from A to B. Hence the execution of A should precede the execution of B. The TDG of Kmeans for a single iteration is shown in Figure 1. Nodes with same color represent multiple instances of the same task. For example, from Listing 7, *subst* task from multiple loop iterations will have the same color. Tasks with no incoming edges indicate that all their data dependencies have been resolved and are scheduled on different worker threads of the SMP. This guarantees the correctness of the application.
In this way SMPSs shifts the burden of identifying data dependencies, movement of data among the processors and scheduling independent tasks to different threads from the programmer to the runtime. The SMPSs runtime only detects data dependences between tasks. Control dependences have to be forcefully applied by the explicit use of synchronization pragmas.

C. Synchronization

In case of control dependencies, the main thread of SMPSs has to be explicitly blocked. For this SMPSs provides synchronization pragmas such as:

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

Listing 5: Task wait pragma

Listing 5 pragma will halt the main thread until the last task updating ”a” has finished execution.

```
#pragma css barrier
```

Listing 6: Barrier Pragma

Listing 6 halts the main thread until all previously generated tasks have been executed. The use of synchronization pragmas has a negative effect on parallelism. For example as mentioned earlier, if functions from Listing 1 are annotated as tasks, a synchronization is required at the end of every iteration of the loop.
To parallelize Listing 1 using SMPSs, the functions are annotated as tasks in the following way:

```
#pragma css task input(newBFD,refCFG) \
      inout(newCFG)
void subst(refCFG, newBFD, newCFG);

#pragma css task input(newCFG,trace) \
        output(dimOUT)
void dimemas(newCFG, trace, dimOUT);

#pragma css task input(newBWD,dimOUT) \
          output(finalOUT)
void extract(newBWD, dimOUT, finalOUT);

#pragma css task input(finalOUT) \
          output(goal_achieved)
void check(finalOUT, goal_achieved);

while(!goal_achieved)
{
   newBFD = pop_queue();
   subst(refCFG, newBFD, newCFG);
   dimemas(newCFG, trace, dimOUT);
   extract(newBWD, dimOUT, finalOUT);
   check(finalOUT, goal_achieved);

#pragma css wait on(goal_achieved)
}
```

Listing 7: SMPSs pseudo code for Listing 1

For each iteration, one instance of these tasks will be added to the TDG. Since the tasks in a single loop iteration are dependent on each other they cannot be executed simultaneously. A *wait* pragma is used at the end of the iteration, to calculate the loop predicate for the next iteration. Since each of the 4 tasks can potentially be executed on different threads, a *wait* pragma is required to evaluate the loop predicate for the next iteration. Figure 2 shows the TDG for Listing 7. Even though multiple iterations are independent of each other, the *wait* pragma at the end of the loop-iteration limits the ability to extract parallelism from the loop. The wait-pragma is a necessity due to the lack of knowledge of the loop-iteration space. The synchronization pragma (wait-pragma) along with dependencies between tasks does not

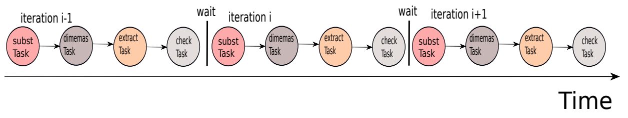

Figure 2: TDG of Listing 7

allow extraction of any parallelism.
In order to avoid this problem, we present a technique of *speculative* generation of tasks from multiple iterations.

III. Speculation in SMPSs

To extract more parallelism from a *while-loop*, we speculatively generate tasks. We achieve this by avoiding the use of synchronization pragmas that block the generation of work. Later the speculatively generated tasks will be validated by the runtime. Figure 3 shows the TDG for Listing 7, when the wait-pragma is avoided. From Figure

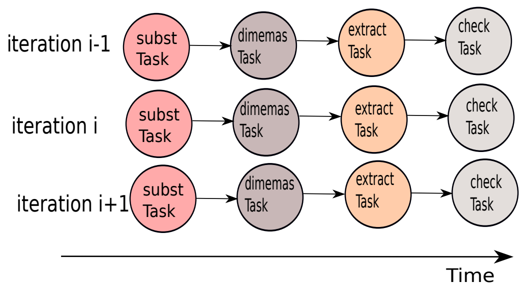

Figure 3: Ideal Speculative TDG of Listing 7

3 we observe an increase in the parallelism extracted from Listing 7. But this is an ideal case. The reason is explained in Section 4.
We describe the terminology that will be used in rest of the paper:

1) Valid tasks - Valid tasks are tasks that are generated even with the synchronization pragma. Skipping the synchronization pragma, does not guarantee the validity of generation of certain tasks. Hence tasks generated from iterations where our speculation succeeds are called valid tasks.
2) Invalid tasks - Tasks generated from iterations where speculation fails are invalid tasks. Updates made by these tasks have to be discarded to maintain correctness.

Consecutive iterations of the loop can either be control or data dependent on each other. If loop iterations are data dependent, then SMPSs runtime will add dependencies between speculative and non-speculative tasks and guarantee the execution of speculative tasks only after the dependencies are resolved. But if the loop iterations are only control dependent, then simultaneous execution of speculative and non-speculative tasks is possible. Since control dependence is enforced using synchronization pragmas, avoiding the pragma would lead to independent tasks being speculatively added to the TDG. This leads to generation and simultaneous execution of *speculative* tasks along with *non-speculative* tasks.
Speculative execution of loop-iterations implies that the main thread of SMPSs generates tasks from multiple iterations without a check of the loop predicate. Hence before the results of such speculatively generated tasks are committed, their validity needs to be ascertained by the worker threads executing these tasks. The information regarding the evaluation of validity of such speculative tasks, has to be made available to the worker threads executing these tasks. Also, if the evaluation fails, i.e., if the speculative tasks are invalid, updates made to the memory by such tasks have to be undone.
To implement the idea of speculative task generation in SMPSs, two of the main issues to be tackled are:

1) Evaluating the validity of tasks - How to propagate information required to evaluate the validity of speculatively generated tasks to worker threads?
2) Rollback of invalid tasks - How to undo results and rollback the changes made by the invalid tasks ?

The following two sections tackle the challenges mentioned above.

A. Pragma for speculation in SMPSs

A new pragma has been added to the SMPSs compiler to annotate a *while-loop* as speculative. This directive states that the tasks from the loop that follows this pragma will be speculatively generated and if possible executed ahead in time. To annotate a *while-loop* as being speculative, a new directive is added to the SMPSs framework. The *speculate* pragma along with its clauses is shown below:

```
#pragma css speculate values(x) wait(y)
```

Listing 8: Speculation Pragma

The clauses for this pragma are:

- *values*: contains parameters of tasks which are called from inside the while loop. The values of these parameters must be protected from invalid tasks. Instead of guarding all the updates performed by these tasks, the programmer can indicate the task parameters whose values need to be protected. Such parameters are passed to the *values* clause.
- *wait*: contains variables whose values determine the continuation of the loop.

By using the speculate pragma, Listing 7 can be transformed into:

```
#pragma css speculate values(finalOUT) \
             wait(goal_achieved)
while(!goal_achieved)
{
   newbd = pop_queue() ;
6     subst(refCFG, newBFD, newCFG);
7     dimemas(newCFG, trace, dimOUT);
8     extract(newBWD, dimOUT, finalOUT);
9     check(finalOUT, goal_achieved);
10  }
```

Listing 9: Speculative loop for Listing 7

As observed in Listing 9, there is no *wait* pragma at the end of the loop-iteration. The *speculate* pragma will generate tasks from multiple iterations of the loop but will check for their validity before committing their results. The information in the *finalOUT* will be protected by updates from invalid tasks since it has been passed to the *values* clause (line 1 Listing 9). The values of *goal_achieved* will be used to verify the validity of speculatively generated tasks (line 2 of Listing 9).

IV. IMPLEMENTATION

To implement the idea of speculative task generation and execution in SMPSs, the existing compiler and runtime of the SMPSs framework are modified. The compiler transforms the input code in order to convey the information regarding two important aspects of *speculative* task generation and execution to the runtime:

1) Evaluating the validity of a *speculative* task.
2) Rolling back of updates done by invalid tasks.

The compiler marks the speculative tasks with special flags to differentiate them from regular tasks. The runtime uses this information to achieve *speculative* execution of tasks.

A. SMPSs compiler modifications

The SMPSs compiler was extended to include the *speculate* pragma along with its clauses, *values* and *wait* (Listing 8). In order to evaluate the validity of a *speculative* task, the SMPSs worker thread executing this task should either be provided with the value of the loop-predicate corresponding to its respective iteration or should have the necessary information to evaluate this value. We use the latter technique.
The SMPSs compiler generates a *guard* function when it encounters the *speculate* pragma. This function evaluates the loop-condition predicate. The function is composed of a single statement, namely the expression from the loop condition. For example, the guard function for Listing 9 is:

```
int guard (void *speculate_params[1])
{
  return (!*(speculate_params[0]));
}
speculate_params[0] = &goal_achieved ;
```

Listing 10: Guard function for Listing 9

Threads executing the speculative tasks use this function to evaluate the validity of the tasks.
The compiler parses and analyzes the expressions used in the loop predicate. It then adds addresses of each of these parameters to the *speculate_params* structure. This structure and the *guard* function are passed as additional parameters to the task. In Listing 10, we observe that the *speculate_params* contains a single element namely the address of *goal_achieved* . The worker thread will pass the *speculate_params* as a parameter to the *guard* function and evaluate the validity of the speculative task. This is the transformation done by the SMPSs compiler to the application code to assist the runtime in evaluating the validity of the speculative task.
The SMPSs compiler marks the task parameters with special *flags*. Based on the clauses the task parameters are marked them with input, output or inout flags. The SMPSs runtime uses this information to analyze the data flow and build the TDG. To aid the runtime in differentiating between parameters passed to the *values* clause and other task parameters a new *speculation* flag is added to the SMPSs framework. The compiler iterates over each of the task parameters annotated inside the while-loop and compares them with variables passed to the *values* clause. If there is a match, it marks the parameter found with a new flag called *css_speculation_flag*. This indicates to the runtime that the updates performed on this variable needs to be protected while executing the speculative tasks. This additional flag is added only to the parameters that are passed into the *values* clause. For example in Listing 9, only the parameter *finalOUT* from *extract* task is be marked with this flag. In this way the compiler transforms the given application code in order to assist the SMPSs runtime in tackling of the issues mentioned at the start of this section.

B. SMPSs runtime changes

The main thread of SMPSs starts executing the thus transformed code by the compiler.

1) Main thread of SMPSs: During execution, the SMPSs main thread detects a task as being a *speculative* task if at least one of its parameters is marked with the *speculation* flag. This implies that a task is *speculative* if and only if one of its parameters is passed to the *values* clause of the *speculate* pragma. The tasks which appear inside the speculative loop but whose parameters need not be protected from invalid instances are not marked as *speculative* tasks. For example in Listing 9, only instances of **extract** are marked as *speculative*. Since the values updated by instances of **subst**, **dimemas** and **check** are not required to be valid at the end of the loop (i.e., when the loop terminates), protecting the updates done by the invalid instances of these tasks would be a waste of resources.
Every iteration of the loop is control dependent on the loop predicate of the corresponding iteration. Hence when the main thread detects a *speculative* task, it adds an input dependency between tasks that access the addresses passed to the *speculate_params* array and the current task. The dependency is added only between the current *speculative*

task and previous instances of the tasks which update the memory locations passed to the *speculate_params* structure. The dependency is required to maintain the validity of the values which are used to evaluate the *guard* function. In Listing 9, to confirm the validity **extract**, we need a valid value of *goal_achieved*, updated inside **check** from the previous iteration. Hence an input dependency is added between *check* task from the previous iteration to the *extract* task of the current iteration. The speculative task is then added to the TDG of the application. Figure 4 shows the TDG of Listing 9 along with its dependencies. If there are data dependencies within or between non speculative and speculative tasks, then the SMPSs runtime honors them by adding edges leading into these tasks in the TDG. This guarantees the correctness of the application. In such cases, the speculative tasks from consecutive iterations are generated but not executed. Even though this leads to only overlapping task generation with task execution, it is an important performance gain since it reduces the runtime overhead incurred due to synchronization.

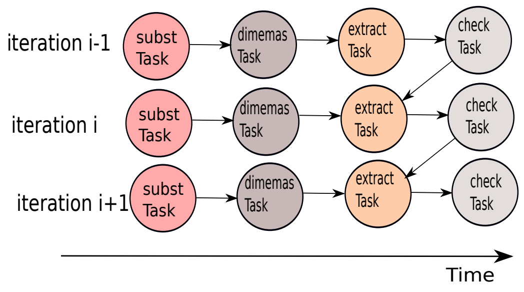

Figure 4: Realistic Speculative TDG of Listing 9

2) Worker threads of SMPSs.: The thread executing the *speculative* task, makes a temporary copy of the parameters marked with the *css_speculation_task* flag. The amount of data to be buffered depends on the size of the parameter. The SMPSs runtime has the information regarding the memory accessed on behalf of each parameter since it uses this information to analyze data dependencies between tasks. For example in Listing 9 a temporary buffer is created for *finalOUT* for each instance of the extract task. The instances of *extract* tasks are then executed. After the execution a check is made using the pointer to the *guard* function. If the check evaluates to true, i.e., the speculation succeeds then the speculative tasks successfully complete its execution. Otherwise the updates done on the parameters passed to the *values* clause are rolled back. If the check evaluates to false, the results in buffers are copied to their actual memory locations. Rolling back in our case involves the copy of the buffer values to the parameters marked with *css_speculation_task* flag due to mis-speculation. The overhead involved in our idea of speculative execution of tasks:

- For valid tasks: one memory copy.
- For invalid tasks: two memory copies, one before the execution and one after execution as our speculation failed.

Also there is an additional overhead involved in evaluating the validity of the speculative task using the guard function. In this way the main thread and worker threads of SMPSs make use of the information provided by the compiler to achieve the *speculative* execution of tasks from loop-iterations ahead in time.

C. Amount of Speculation Allowed

The amount of speculation allowed implies how many iterations should be speculatively generated ahead of time. More speculation implies more generation of work resulting into an increase in parallelism. But this may also lead to an increase in the number of speculatively generated invalid tasks. The trade off is between an increase in parallelism versus time and resources spent on the execution of invalid tasks. The maximum task count in SMPSs is 1000 by default [1]. This implies that the main thread continues to generate tasks until either it is explicitly blocked or the number of nodes in the TDG reaches 1000. Hence if we do not limit the number of *speculative* tasks that are generated, the main thread will generate 1000 tasks, i.e., 1000/*tasks_per_iteration* iterations. Often this may lead to the generation of a large number of invalid tasks. Hence in order to control this, a new environment variable was added to the SMPSs runtime(css_speculation_tasks). By default, the value of this environment variable is 10. This implies that tasks from 10 iterations ahead in time will be added to the TDG. The main thread then stops the task generation until these tasks have been executed and again generates tasks from another 10 iterations in case the loop needs to be continued. The programmer can use this environment variable to control the amount of speculation.

V. Results

We tested our idea on four applications from the domain of linear algebra and graph algorithms. The performance comparison is done between SMPSs version of these applications using the *speculate* clause and without (i.e., the version which uses synchronization pragma at the end of loop iteration).

1) Gauss-Seidel Method - Gauss-Seidel method is a technique for solving n equations of a linear system of equations, Ax=b. The coefficients of the equation are improved in every iteration using the formula, x^{k+1}=D^{-1}(b-Rx^k), where D is the diagonal component of A and R is the remainder. The algorithm runs

[1]This is the SMPSs runtime feature for memory management and to control the size of the graph

in a loop until the absolute approximate error is less than a prespecified tolerance for all unknowns. Hence at the end of every iteration a *wait* is performed to check the convergence. To avoid this a *wait*, *speculate* pragma is used.

2) Jacobi - Jacobi is a classical linear iterative solver which approximates all the unknown variables at a time.
3) Kmeans - In statistics and machine learning, k-means clustering is a method of cluster analysis which aims to partition n observations into k clusters where each observation belongs to the cluster with the nearest mean.
4) Lee-routing - Given a maze, this benchmark finds the shortest-distance paths between pairs of starting and ending points. We present the results of Lee-routing separately since this is the only application in our analysis where in the loop iterations are control dependent.

The above mentioned applications were executed on a 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 were executed with two different problem sizes. Jacobi and Gauss-Seidel were executed with 4096 and 8192 unknowns respectively. Kmeans was executed with 1 million and 10 million data points respectively.

Performance evaluation is done on four major aspects of speculative execution, namely,

1) Speedup - Speedup of the speculative versions of the application
2) Normalized execution - Normalized execution time of the applications compared to their non-speculative version using the formula
   Nt = Tns / Ts, where,
   Tns - time taken by non speculative version
   Ts - time taken by speculative version .
3) Amount of speculation allowed - Iteration window space for speculation.
4) Task wait time - Time spent by threads in waiting for tasks.

A. Speedup

Figure 5 shows the speedup of the speculative version of the Jacobi algorithm with two different problem sizes. The *wait* at the end of every iteration of the loop to check for the convergence is avoided. Tasks from consecutive iterations are data dependent and hence we can only overlap the generation of speculative tasks with the execution of tasks from the current iteration. But by increasing the problem size we achieve a higher performance due to the inherent intra-loop parallelism present in the algorithm. An increase in

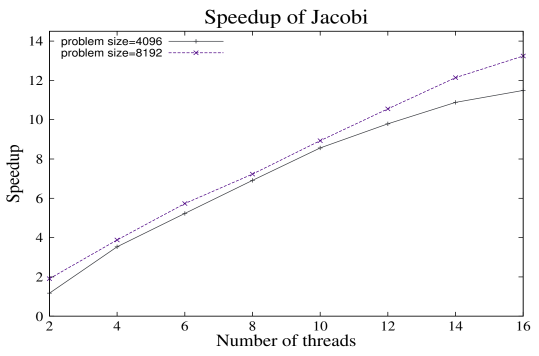

Figure 5: Speedup of Jacobi algorithm.

the problem size implies an increase in the number of tasks generated in each iteration and an increase in the number of speculative tasks added to the TDG. Both of them combined allow effective use of the available resources which leads to a higher speedup with an increase in the problem size. Fig 6

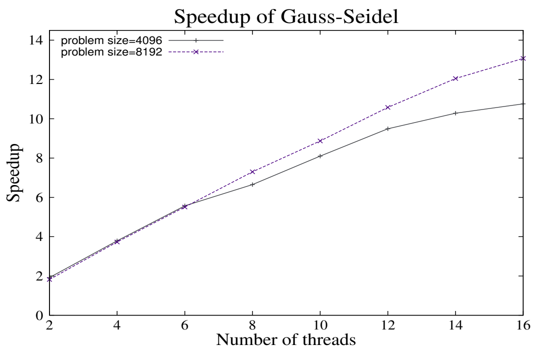

Figure 6: Speedup of Gauss Seidel algorithm.

shows the speedup of the speculative version of Gauss-Seidel algorithm for similar problem sizes. Gauss-Seidel converges faster but does not scale similar to Jacobi. Also similar to Jacobi, tasks from consecutive iterations of Gauss-seidel algorithm are data dependent. With an increase in problem size there are more tasks are speculatively added to the TDG which allows a better usage of available resources. Hence we observe an increase in the speedup with higher number of threads for a larger problem size.

Fig 7 shows the speedup of Kmeans algorithm with 1 million and 10 million data points. In every iteration of kmeans, the cluster centers are updated and the error is calculated. If the error is less than a specified tolerance level then the loop terminates. Tasks from every iteration update on the same block of clusters. Hence tasks from consecutive iterations are data dependent. But since in a single loop, parallel updating of clusters can be done, we obtain a lot of intra-loop parallelism which is seen in the figure since both the problem sizes scale similarly. Speculative tasks are only added to the TDG and executed only when possible.

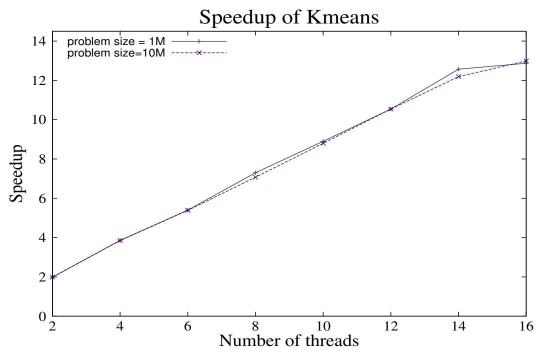

Figure 7: Speedup of Kmeans algorithm

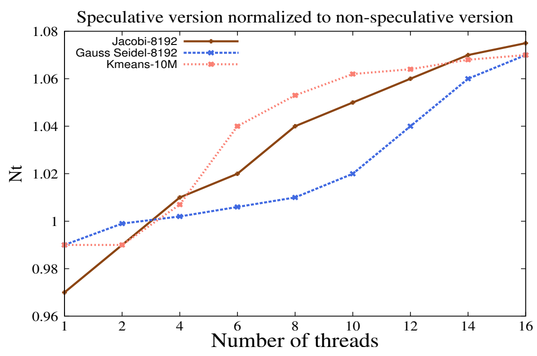

Figure 9: Normalized Speculative execution.

B. Normalized Results

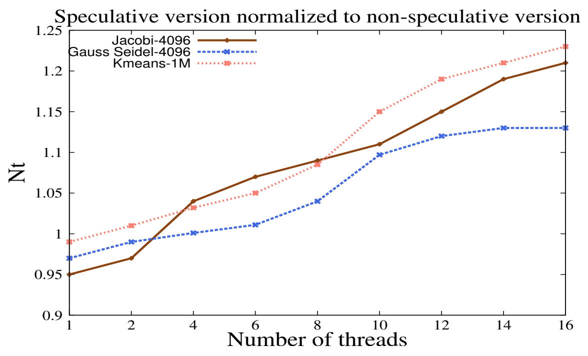

Figure 8: Normalized Speculative execution.

Figure 8 shows normalized time of speculative execution to non speculative execution for smaller problem sizes. All three applications achieve higher performance with increasing number of threads. The non speculative versions of the applications generate enough parallelism to occupy smaller number of threads . But with an increase in the number of cores there are more execution resources to execute speculative tasks. Hence speculative execution gains with increasing number of threads. Jacobi and Kmeans offer intra-loop parallelism to a higher degree which allows even the non-speculative versions of their algorithms to scale. With 16 threads Jacobi and Kmeans show an increase of 1.21x and 1.26x respectively when compared to their non-speculative versions. In comparison Gauss-Seidel gains only 1.14x. But still it is a significant improvement since the comparison is being done with a parallel version of the algorithm. Figure 9 shows normalized time of speculative execution to non speculative execution for larger problem sizes. Comparing Fig 8 and Fig 9, we observe a drop in the performance. The applications chosen contains intra-loop parallelism even without speculation. By increasing the problem sizes, there is an increase in the amount of parallelism that can be extracted from each iteration of the loop. This reduces the effect of speculative execution since the parallelism present in a single loop-iteration increases. But the graph also shows that with increasing number of threads we can achieve better performance even with higher problem sizes.

1) Lee-routing: We analyze the lee routing algorithm separately since this is the only algorithm in our chosen set of applications where the loop iterations are control dependent. The algorithm has two phases, expansion and traceback. The idea of speculation is applied in the expansion phase. In this phase, the algorithm searches for a shortest path between the start and end points by performing a Breadth First Search. Every iteration of this phase explores certain points and checks for the end point. Hence it is possible to parallely execute tasks from multiple iterations. The speedup shown in Figure 10 is only for the expansion phase.

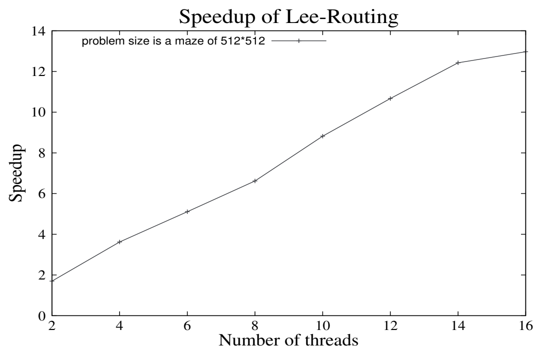

Figure 10: Speedup of Lee-Routing algorithm.

Figure 11 shows the normalized execution time of speculative lee routing algorithm to non-speculative version. We observe an improvement of 12x with 16 threads in this case. Since tasks from multiple iterations can be simultaneously executed, we gain the maximum from this application with the idea of speculation.

C. Amount of Speculation Allowed

In Figure 12, we compare the speedup when the value of css_speculation_tasks varies from 20 to 100. The results shown in this Figure are from running the applications with 8 threads for smaller data sizes. From the figure we

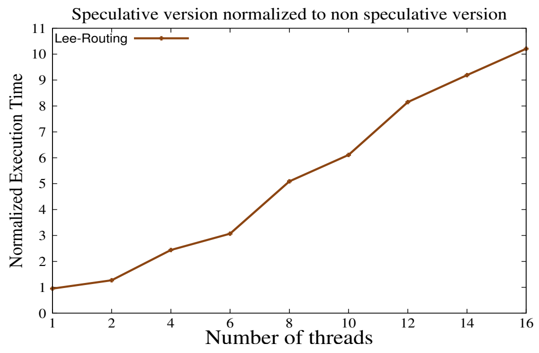

Figure 11: Normalized speedup of Lee-Routing algorithm.

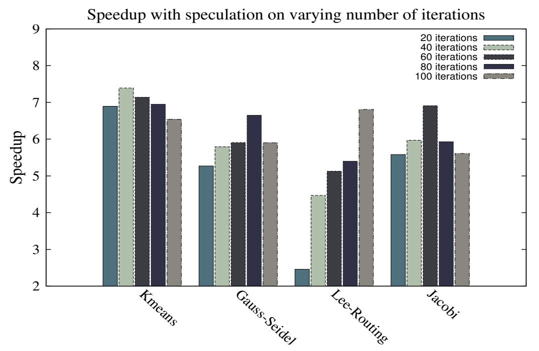

Figure 12: Comparing iteration space.

can observe that by increasing the window of iteration space performance increases. But after a threshold, the overhead incurred due to the invalid tasks overshadows the performance benefits. Invalid tasks create an overhead and hence have an effect in the speedup performance. Figure 12 also shows that the threshold for performance dip with varying iteration spaces is specific to the application. Hence we observe different optimal values css_speculation_tasks for different applications.

D. Taskwait time by Threads

Speculative generation of tasks implies increase in the number of tasks added to the TDG of the application. Since task generation is not blocked due to the synchronization pragmas, more parallelism is extracted from the input code. This is true irrespective of the type of dependencies (control or data) between or within tasks from speculative and non-speculative iterations. This leads to a decrease in time spent by threads waiting for tasks. We analyzed such effects of speculative task generation on the number of tasks generated and the corresponding effect on task wait time by threads.
For this profiling we used Paraver[1], a flexible visualization tool to analyze the characteristics of *speculative* task generation. Figure 13 shows a histogram comparing the wait times of speculative and non speculative versions of the three applications executed on 8 threads each. The Y axis

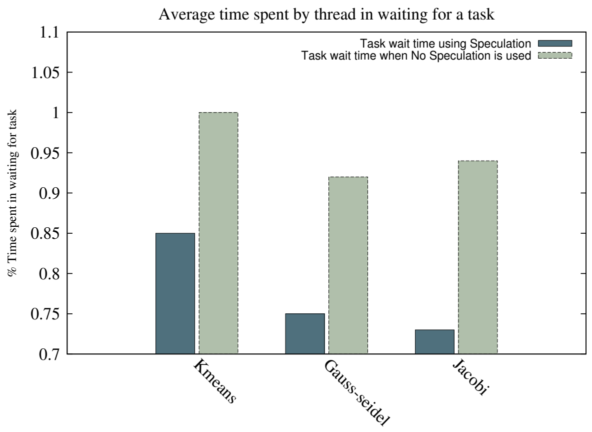

Figure 13: Time spent by threads waiting for tasks.

Table I: Tasks Generated

Application	% increase in tasks
Jacobi	13%
Gauss-Seidel	7.83%
Kmeans	10.92%
Lee-Routing	37%

shows the average execution time spent on waiting for tasks by threads. Although by speculative execution we achieve 15-25% reduced time spent by threads in waiting for tasks, similar improvement is not reflected in the performance. The reason being, more number of tasks does not imply their parallel execution. The latter is the characteristic of the parallelism offered by the TDG. *Speculative* generation of tasks implies generation of invalid tasks, whose updates are not committed. This implies their execution is overhead which does not improve performance.

E. Overhead of Invalid Tasks

As discussed in earlier sections, generation and execution of invalid tasks is an overhead which is not present in the non-speculative version of the application. Table 1 compares % increase in speculative tasks when compared with non speculative version. The data shown is when applications are run on 8 threads. But this overhead is acceptable since we gain performance benefits by speculatively generating tasks ahead in time.

VI. Related Work

L.Rauchwerger and D.Padua have proposed a technique to parallelize *while-loops* which involve linked-list traversal in [13] . In this paper, the authors present a framework for automatic transformation of while-loop if the remainder is parallel. In [12], techniques to obtain vector like performance on multiple issued pipelined processor has been proposed.
OpenMP [5] is a shared memory programming model

that supports multiprocessor programming in C, C++ and Fortran. Cilk [6] is another such programming model for algorithmic multithreaded programming developed at MIT. Although OpenMP and Cilk are both task based and loop based programming models, they do not support dependency aware task execution. The idea of *speculating* in a task based programming model has been explored in [3]. The authors of this paper have proposed the idea of branch prediction and value speculation as techniques to skip synchronization points in branches and loops in OmpSs [4].
The idea of executing on copies of data has been explored in [8]. The authors of this paper execute the iterations on copies of data and later commit the results. We execute the tasks on the actual data and rollback when the speculation fails. Ours is a more optimistic way of execution.

VII. Conclusions

The sequential execution of *while-loop* iterations heavily restricts parallelism. With multi-core processors becoming the norm in today's computing, the inability to parallelize such frequently used constructs is more prominent. In order to overcome this problem, the idea of *speculative* generation and possible execution of loop-iterations ahead in time is proposed in this paper. We optimistically predict on the execution of the future iterations of the loop. This optimism arises from the fact that in practice such loops execute for multiple iterations before terminating.
We implement our idea in SMPSs, a task based dataflow programming model for Symmetric Multiprocessors. By use of this idea, we gain an average speedup of around 1.2x while executing the applications with 16 threads, for loops where the consecutive iterations are data dependent.

VIII. Future Work

Currently *speculatively* generated tasks are executed irrespective of their validity and later rolled back in case the speculation fails. This will lead to invalid tasks being executed even if their validity has failed. In the future we plan to optimize the *speculation* idea by implementing a mechanism which will propagate the results across speculative tasks in case the speculation fails.
Another important optimization is to generate *speculative* tasks depending on the size of the data that needs to be protected. Since making copies of data is an overhead in our idea of *speculation*, we would like to find the threshold where this overhead starts degrading the performance.

IX. Acknowledgments

We thankfully acknowledge the support of the European Commission through the TERAFLUX project (FP7-249013) and the HiPEAC-3 (contract FP7-ICT 287759) Network of Excellence (FP7/ICT 217068), the support of the Spanish Ministry of Education (TIN2007-60625, CSD2007-00050 and FI program), the Generalitat de Catalunya (2009- SGR-980) and the national project (TIN2012-34557).

References

[1] http://www.bsc.es/media/1364.pdf.

[2] http://www.iiserpune.ac.in/ pgoel/gaussseidel.pdf.

[3] N. Azuelos, Y. Etsion, I. Keidar, A. Zaksy, and E. Ayguade. Introducing speculative optimizations in task dataflow with language extensions and runtime support.

[4] A. Duran, E. Ayguade, R. M. Badia, and J. Labarta. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters, Volume 21, Number 2, p.173-193 (2011)*, pages 173–193, March 2011.

[5] A. Duran, J. M. Pérez, E. Eduard Ayguadé, R. M. Badia, and J. Labarta. Extending the OpenMP Tasking Model to Allow Dependent Tasks. In *OpenMP in a New Era of Parallelism*, pages 111–122. Springer Berlin / Heidelberg, 2008.

[6] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *In Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '98, Montreal, Canada, 1998. ACM.

[7] R. K. Gayatri, R. M. Badia, and E. Ayguade. Transactional access to shared memory in starss, a task based programming model. Europ-par, Rhodes Island, Greece, 2012. Springer Berlin Heidelberg.

[8] C. T. M. F. V. N. R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '2008, Montreal, Canada, 2008. IEEE.

[9] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Morgan and Claypool Publishers series, 2nd edition, 2010.

[10] J. M. Perez, R. M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. *IEEE Int. Conference on Cluster Computing*, pages 142–151, September 2008.

[11] J. M. Perez, R. M. Badia, and J. Labarta. Handling task dependencies under strided and aliased references. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 263–274, New York, NY, USA, 2010. ACM.

[12] L. Rauchwerger and D. Padua. Parallelizing while loops for multiprocessor systems. In *IN PROCEEDINGS OF THE 9TH INTERNATIONAL PARALLEL PROCESSING SYMPOSIUM*, 1995.

[13] L. Rauchwerger and D. A. Padua. Parallelizing while loops for multiprocessor systems. In *Proceedings of the 9th International Symposium on Parallel Processing*, IPPS '95, pages 347–356, Washington, DC, USA, 1995. IEEE Computer Society.