Robust Thread-Level Speculation

Álvaro García-Yágüez, Diego R. Llanos, Arturo González-Escribano
Dpto. de Informática
Univ. de Valladolid, Spain
Email: alvarga87@gmail.com, diego@infor.uva.es, arturo@infor.uva.es

Abstract

Robustness is a key issue on any runtime system that aims to speed up the execution of a program. However, robustness considerations are commonly overlooked when new software-based, thread-level speculation (STLS) systems are proposed. This paper highlights the relevance of the problem, showing different situations when the use of incorrect data can irreversibly alter the speculative execution of an algorithm, despite the efforts of a given STLS system to maintain sequential consistency. We show that the management of speculative exceptions is a common factor to these problems. Based on this fact, we propose a novel solution to handle speculative exceptions. Our solution eagerly tries to solve the issue before the non-speculative thread arrives to the instruction that rose the exception. We compare our solution to a more conservative approach found in the bibliography. The comparison is done both qualitatively, through a detailed analysis of the tradeoffs involved, and quantitatively, evaluating the effects of both solutions in the execution of three different benchmarks on a real system. Both studies conclude that our solution handles the occurrence of speculative exceptions more efficiently. Under heavy loads intended to push to its limits a STLS system, our solution leads to execution times reduced by up to 52.02% with respect to earlier proposals. Our solution does not affect the performance when speculative exceptions do not appear. We believe that our proposal makes STLS systems robust enough to be used in production environments.

1. Introduction

Software Thread-Level Speculation (STLS) has empirically demonstrated that it is a suitable solution for the parallelization of applications that cannot be analyzed at compile time. Several research groups are committed to get results using speculative approaches in order to parallelize benchmarks applications taken from SPEC2000 (see e.g. [1], [2], [3]), SPEC2006 (see [2]) and others irregular applications [4], [5], [6].

In essence, STLS consists in the optimistic, parallel execution of tasks originally intended to be carried out sequentially. These tasks are usually iterations within a loop, although there are mechanisms that allow the parallelization of other parts of sequential code as well (see e.g. [2]). Without loss of generality, we will focus our discussion in loop-level speculative parallelization. With STLS, tasks are composed by several, consecutive iterations, called "blocks". These blocks are optimistically executed in parallel, assuming that no dependencies will occur among them. If a dependence violation appears, affected tasks should be restarted to ensure that they are running with correct data. Therefore, results calculated so far will either be discarded if a separately memory storage space is employed [4], [5], [3], [2] or rolled back, recovering the values stored before running these tasks [6].

STLS solutions are steadily becoming more mature. This makes possible the use of these techniques in complex applications and environments, even in the presence of dynamic data structures [7]. But despite all the work made so far in the field, it is a far path yet to consider STLS as a global solution to the problem of automatic parallelization. Unfortunately, most of the proposed solutions does not consider all of the implications of dealing with potentially-incorrect, speculative data. Suppose that a speculative thread consumes a datum incorrectly set to zero. Before the software-based monitor that controls the speculative execution detects the dependence violation and performs the corrective actions, the offending thread can use this datum as a divisor in an arithmetic operation, leading to a segmentation fault that makes the entire application to crash. We will call *polluted data* these incorrect, speculative data. Other real-world problems that may arise include the use of polluted pointers or polluted indexes to access shared data, unexpected `breaks` that suddenly stop the parallel execution, or even endless loops.

This work examines in detail such situations, identifying a common factor to all of them and proposing a solution that is tested in a real environment. The contributions of this paper are the following:

- We study in detail some problems that current, still-experimental STLS systems should solve before STLS could be used in a productive environment. These problems include the management of speculative exceptions, how to prevent the incorrect modification of shared-memory locations that can not be undone, the unexpected end of speculative sections, and the fall into endless loops due to the use of polluted data. We

978-1-4577-1950-9/11/$26.00

isolate the management of speculative exceptions as responsible of several of these problems.
- We compare two different alternatives to handle speculative exceptions: A more conservative approach proposed in [7], with a new, "eager" approach based on immediate corrective actions. We also examine the tradeoffs involved from a qualitative point of view, concluding that our proposal overcomes earlier approaches in terms of performance.
- We extend a software-based speculative engine [8] in order to apply both solutions to solve the occurrence of speculative exceptions **on a real system**. Our experimental results shows that our solution leads to up a 52.02% reduction of the running time over a more conservative attempt to solve this problem.

The result is a more robust scheme for software-based speculative parallelization, that can not only be used to execute a handful of benchmarks in laboratory conditions, but also to be used in a production environment.

The rest of this paper is organized as follows. Section 2 introduces STLS. Section 3 explains some problems that parallel speculation should take into account to broaden the number of applications to deal with, and shows that the management of speculative exceptions is behind several of them. Section 4 describes the solution space to the speculative exceptions problem. Section 5 analyzes in detail the tradeoffs of two solution to this problem. Section 6 shows experimental results on a real system to compare both approaches. Section 7 discusses some related work, while while Section 8 concludes the paper.

2. Software Thread Level Speculation

Software Thread-Level Speculation (STLS) aims to speculatively execute in parallel fragments of code without the need of a prior, compile-time analysis to detect potential dependence violations. The code (typically loop iterations) are divided into blocks and assigned to threads. The sequential semantics impose a total order among these blocks. At a given moment, there is a *non-speculative* thread that is executing the earlier block, and *speculative* threads executing the remaining blocks. Note that only speculative threads might consume an incorrect, speculative datum generated by a predecessor. Thus, only speculative threads might need to be squashed if a dependence violation appears.

There are two different approaches to handle speculative data. The first one is the use of version copies, one per thread. With this approach, if a thread successfully finishes the execution of its block of iterations, its version copy is committed to the main copy of the shared data. Note that, to maintain sequential semantics, commits should be done in order, from the non-speculative to the most-speculative thread. If a dependence violation appear, incorrect version data is simply discarded. This technique is used in several solutions found in the literature (such as [4], [5], [3], [2]). The second solution to handle speculative data is to perform changes directly in the main copy of the shared data speculatively accessed. In this case, it is necessary to keep track of the changes being made in order to roll them back if a dependence violation appear [6]. The speculative scheme used in this paper is based on [4], that uses the former solution.

In order to track potential dependence violations, there are again two different solutions. The first one, adopted by the speculative scheme used, defines a set of flags to mark each datum as speculatively accessed. With this solution, in the case of a *speculative load*, a given thread searches the most up-to-date version of the datum needed by simply accessing to the version copies owned by its predecessors and checking the corresponding flags to see if the datum is present there. On the other hand, in the case of an *speculative store*, the thread should check if any successor has *already* consumed an incorrect value of this datum. Again, this check can simply be done by accessing to the corresponding flags of the versions owned by each successor. Note that these operations can be carried out without other threads were aware of the situation.

The second solution to track dependence violations is to use signals to synchronize the access of each thread to data owned by other threads [9]. Note that, unlike the previous approach, this solution needs the cooperation of several threads to solve a particular request, while in the previous solution both the forwarding operation and the search for violations can be done without any explicit synchronization mechanism.

3. Weaknesses of current STLS proposals

In TLS systems, the access to speculative data should be monitorized in order to avoid inconsistencies with respect to sequential semantics. In software-based TLS, this task is carried out by a software monitor, responsible of ensuring that all threads consuming polluted data are eventually squashed and restarted with correct values. Nevertheless, as we will see below, there are several situations where this corrective action may happen too late. These situations, that were mostly overlooked by the research community, prevent in fact the use of existent software-based speculative schemas in production runs.

- **Speculative exceptions.** The use of a polluted datum as an operand in arithmetic operations can produce Float Point Exceptions (FPE), as well as Segmentation Fault Exceptions (SFE) when used in memory operations. Note that exceptions due to the use of polluted data would not have happened in a sequential execution of the algorithm. We believe that the possibility of occurrence of such a situation is a reason strong enough

```
1  x = z * mainCopy[20];
2  i=0;
3  while( x < 100)
4  {
5    x = x * z;
6    i++;
7  }
```

(a)

```
1  x = z * specload(mainCopy,20);
2  i=0;
3  while( x < 100)
4  {
5    x = x * z;
6    i++;
7  }
```

(b)

Figure 1. Endless loop due to a polluted data. (a) Original code. (b) Speculative code.

to adding speculative exceptions support to any STLS system.

- **Out-of-bound accesses to shared memory.** The use by a thread of a polluted datum as an index or pointer to store data may lead to the incorrect modification of the thread's version copy of shared data. Such changes can be reverted discarding this version copy and re-starting the thread with correct values. However, if no bound checking is performed, data could be incorrectly written to version copies owned by other threads, a write operation that could not be reverted. Note that this problem may arise without leading to a Segmentation Fault exception, because languages such as C/C++ allows the program to access data beyond the boundaries of the original structure (the "buffer overflow" problem). This problem is particularly relevant in shared-memory STLS implementations, where accesses beyond speculative data boundaries should be controlled, even at the cost of a performance degradation. Although distributed memory implementations of STLS can ignore this problem, most STLS solutions proposed so far relies on shared memory. If the STLS system supports speculative exceptions, one possible solution to this problem is use this mechanism to artificially raise an exception if the boundaries of the structure intended to be updated are crossed.
- **Unexpected end of speculative sections.** Instructions such as break, return, or exit calls can appear in the body of a speculative loop. The execution of these statements affects to the parallel execution of the entire loop and can not be easily undone. The use of polluted data may lead a speculative task to an incorrect state, executing one of these statements before being notified about the dependence violation. Such statements would finish the speculative execution, thus violating sequential semantics. We have discovered that, if the STLS system used support speculative exceptions, statements that lead the parallel execution to a non-reversible state may be instrumented, in order to raise an exception that either freezes the speculative thread until it becomes non-speculative or re-start the task in the hope that updated values will not lead to such a situation.
- **Falling into endless loops.** Figure 1 shows an example of this problem. Suppose that `x` and `z` are private variables of a speculative task and `specData` is a shared structure being speculatively accessed. Figure 1(a) shows the original code, while Fig. 1(b) shows the code after replacing the original read to the structure with a `specload()` call. If this function call speculatively returns a value equal to 0, the task will enter into an endless loop. Recall that speculative tasks do not monitor continuously their own state, but only when speculative functions such as `specload()` or `specstore()` are called. Therefore, our task will have no way to attend the squash operation that the predecessor task will issue as soon as the dependence violation is discovered. There are two possible solutions to this problem: To insert additional speculative calls inside the `while` loop to make the task continuously monitor its own execution status, with the subsequent performance degradation, or to find a way to stop and restart the task from outside.

These four situations have several factors in common. All of them take place when a polluted datum is used before the dependence violation could be notified to the consumer task. This use of a polluted datum changes the execution flow and/or the state of the entire application without any possible recovery. Such situations have been overlooked by the research community, mostly because these situations are hard to find, particularly when only a handful of benchmarks is used to test speculative parallelization solutions. However, a STLS scheme intended to be used for production purposes should take all of them into account.

As we saw before, some of these situations may be managed if the STLS system incorporates support to speculative exceptions. In the following section we will study how to solve this problem.

4. The Speculative Exceptions problem

As we stated in the previous section, as soon as a dependence violation is detected, the offending tasks will be squashed and restarted. However, until the violation is discovered, tasks that have consumed polluted data could have executed operations that leads to situations that would

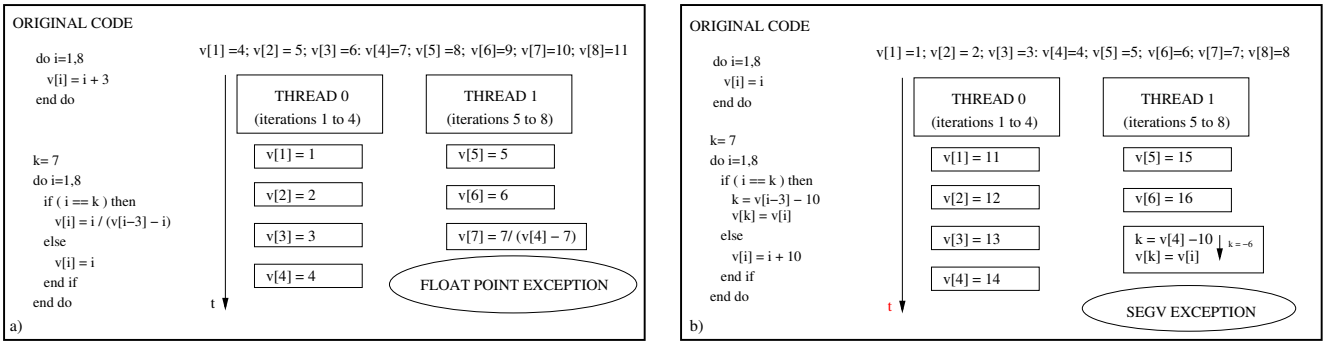

Figure 2. Code examples that produce exceptions when they are speculatively parallelized. Floating Point Exception (a) and Segmentation Fault Exception (b).

not have happen in the original, sequential code. From here on, we will focus on the management of speculative exceptions, a problem that is in the core of most of the situations described in the previous section.

Figure 2 shows two situations that can raise an exception in a speculative, parallel execution. Code a) produces an arithmetic exception, while code b) generates a segmentation fault caused for an access outside the boundaries of array v. For each situation, the original code is depicted at the left of the figure. In both cases, the code shown first initializes the vector to be speculatively accessed, and then a loop operates with that vector. This second vector is the one that will be speculatively parallelized. For simplicity, only the sequential version of these loops are shown.

Suppose that two threads cooperate executing the second loop. Each thread receives a block of four consecutive iterations: Thread 0 executes iterations 1 to 4, while Thread 1 executes iterations 5 to 8. Therefore, Thread 0 is the non-speculative thread, while Thread 1 is the most speculative one. The evolution is similar for both examples. When Thread 1 runs iteration 7, it speculatively reads $v[4]$. At that moment, and supposing a perfect synchronization without loss of generality, Thread 0 have not yet updated $v[4]$. Therefore, the most up-to-date value for $v[4]$ is fetched from the vector main copy. Before Thread 0 speculatively writes a new datum on $v[4]$ and detects that Thread 1 has consumed an outdated value, Thread 1 has executed an operation that raises an exception (a division by zero in the left figure, and an access to element with index -6 in the right figure).

Let us highlight the importance of the problem. If no action is taken and an exception is raised, the entire application will crash, not only the speculative section. Even worse, if the incorrect index being accessed in Fig. 2, right, is a positive value instead of -6 that lies outside the boundaries of the version copy owned by Thread 1, the segmentation violation exception might not raise, but the result of the speculative execution would differ with respect to the sequential results.

It is not easy to handle such situations, but any speculative scheme that aims to be used in a production environment should give an answer to this problem. To ignore this possibility in the hope that it is a very uncommon situation may be acceptable for prototyping speculative schemes, where a handful of well-known benchmarks does not lead to such behaviors, but not if STLS is aimed to be integrated into existent parallelization compilers and runtime support.

We can distinguish four different solutions to this problem, from the most conservative to the most aggressive one.

1) To avoid the use of STLS in loops that contain operations that can potentially produce exceptions. This solution restricts the code that can benefit from this technique. Moreover, it is not clear whether such an analysis could be carried out only with compile-time information.
2) To allow the use of STLS in these cases, but instrumenting the code with functions that stall a speculative thread *before* executing an operation that *could* produce an exception. The execution will be stalled until the thread is transformed into non-speculative. This solution prevents the occurrence of speculative exceptions, at the cost of highly reduce the performance of applications. Moreover, the difficulties of detecting such potentially dangerous operations at compile time still remains.
3) The third solution is to handle the exception, and make the signal handler to stall the thread until it has become the non-speculative one. This solution has been recently proposed by C. Tian *et al.* [7]. Once reached the non-speculative state, this solution allows to distinguish between exceptions due to the speculative execution and those due to an error in the original code. However, as we will see, it comes at the cost of a noticeable performance degradation.
4) Finally, the most eager solution is to handle the exception and take corrective actions as soon as it arises. As far as we know, such a solution has not been proposed in the literature. Our proposal is to squash the offending thread (and it successors) in order

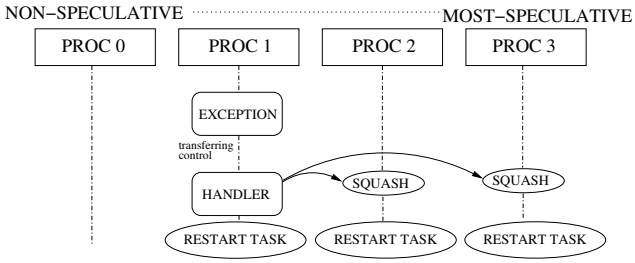

Figure 3. The "InclusiveSquash" solution: Squashing tasks when an exceptions arises on a speculative thread.

to restart them with updated values. As we will see in the following sections, this behavior is simpler to describe than to implement, because runtime exceptions are handled by the same processor that raises the exception. If the exception raises again, the squashing and restarting process is carried out again. This leads to two different scenarios: (a) the dependence violation that was producing the exception is finally solved, or (b) the thread becomes non-speculative. In this last case, if the exception raises again, then the problem is due to the original code and not to the STLS system.

As we will see in Sect. 6, our proposal noticeably improves the performance of Tian *et al.* solution, cutting the execution time to one-half in some extreme cases. In the following sections we will describe how to deal with the distributed management of speculative exceptions in the context of a state-of-the-art STLS system.

5. Handling speculative exceptions

As we stated in the previous section, there are two different possibilities to be considered in order to manage speculative exceptions. The first one, that we will call "SpinWait", stalls the execution of the thread until it becomes non-speculative. The second one, that we will call "InclusiveSquash", takes corrective actions as soon as the exception arises, by means of a handler that decides the next action to be taken. If the handler decides to squash the task, the offending thread that have risen the exception is squashed and restarted, together with *all* its successors (see Fig. 3). This behavior gives the solution its name.

5.1. Exceptions due to the use of polluted data: Tradeoffs

We will now briefly consider the tradeoffs of both solutions in terms of performance. Suppose we have two processors, P0 and P1, speculatively executing a loop in parallel. The loop is divided into four blocks of iterations, that we will call tasks T0, T1, T2 and T3. Figure 4 describes

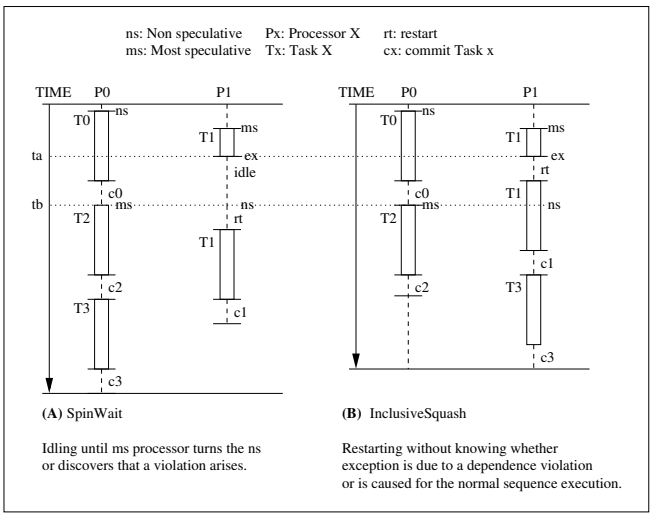

Figure 4. Exceptions due to the use of polluted data.

both possible behaviors when dealing with a speculative exception that raise due to polluted data. Suppose that P0 starts the execution of T0, therefore acting as the non-speculative thread, while P1 starts with the execution of T1. At a given moment, labeled *ta* in the figure, P1 raises an exception.

The "SpinWait" solution (see Fig. 4, left) stalls P1 until the dependence violation is notified. If dependence violation checks are done at commit time, P1 will be idle until P0 has (a) finished task T0, (b) committed results (c) notified P1 about the dependence violation, and (d) started task T2. Note that this notification arrives to P1 at the same time than the non-speculative label for task T1 (instant *tb*). On the other hand, if the speculative scheme being used notifies the dependence violation when performing speculative stores (default behavior in [8]), the restart might be produced slightly earlier. In both cases, after some time devoted to restart the task (labeled as *rt*), processor P1 starts the re-execution of T1. If the exception was due to the use of polluted data, the problem will not appear again, since T1 has no running predecessors that might update data values. Finally, after executing task T2, P0 would start the execution of T3, the task that had not been assigned yet.

We now briefly discuss how the "InclusiveSquash" solution (Fig. 4, right) would act in this case. Once the speculative exception has been risen, this solution immediately stops task T1, restarting it after some time needed to clean the corresponding data structures (*rt*). If we suppose all tasks of equal duration, T2 would then be assigned to P0. As can be seen, this second solution would lead to a performance improvement over the previous scheme. Our experimental results, shown in Sect. 6, confirm this forecast.

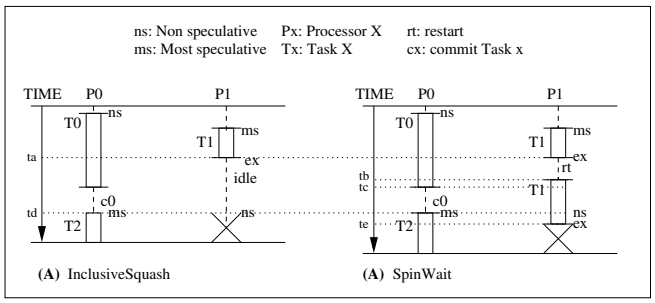

Figure 5. Exceptions due to the original application.

5.2. Exceptions due to the original code: Tradeoffs

After reviewing how both solutions deal with exceptions due to polluted data, let us examine what would happen if the exception is produced by a failure in the original application.

The "SpinWait" solution (Fig. 5, left) will stall task T1, deferring its re-execution until either a dependence violation is notified or T1 becomes non speculative. Since the speculative execution has nothing to do with the exception, the dependence violation notification never arrives. When T1 becomes non speculative (time *td*), the task is re-started and the application crashes, reporting the error to the operating system.

On the other hand, the "InclusiveSquash" solution immediately restarts the execution of the speculative task (fig. 5, right). The exception will raise again, one or several times, until T1 has become the non-speculative task. Interestingly, the first time the exception raises being T1 the non speculative task (instant *te*), the reason may still be a dependence violation produced in the "speculative phase" of task T1 (time elapsed between *tb* and *td* or an exception due to the original code. In the first case, T1 has already been marked as squashed, so the task will be restarted once again. Otherwise, the application is allowed to crash. Note that the extra time needed by the "InclusiveSquash" approach is not relevant in this case, since the application will crash only once.

Taking everything into account, we can conclude that "SpinWait" informs faster about an error when an exception is produced because of the original sequence, whereas "InclusiveSquash" obtains better results when the exception is due to a dependence violation. Since ensuring a consistent parallel execution is far more important than spending some more time when an application crashes due to an exception, we can conclude from this qualitative study that the best mechanism to control exceptions in STLS is "InclusiveSquash". As we will see, experimental results confirm this fact.

6. Experimental evaluation

In this section we perform an experimental evaluation of the two proposed solutions to the speculative exceptions problem.

6.1. Implementation details and OpenMP issues

In order to quantitatively evaluate the design space of the solutions to the speculative exception problem, we have augmented a state-of-the-art speculative scheme first presented in [4] with support to handle speculative exceptions. The entire STLS scheme has been implemented using C and OpenMP 3.0.

Figure 6 shows a flow diagram describing what actions should be carried out to handle exceptions using the "InclusiveSquash" scheme. The diagram is based on [10] patent which explains how to handle exceptions in ILP processors with speculative operations. As can be seen, exceptions are reported only if the task is non speculative and no dependence violations have appeared. Otherwise the system assumes that the exception is produced by a dependence violation and therefore will be processed according to the behavior taken when a violation appears as Figure 3 depicts.

To understand how we handle speculative exceptions with this STLS scheme, it is necessary to take a look at the OpenMP standard. OpenMP v3.0 specifications (p. 45) [11] states that "A throw executed inside a loop region must cause execution to resume within the same parallel region, and the same thread that threw the exception must catch it." According to this statement, when an exception appears, the execution control is transferred to an exception handler that will be executed by the same thread that provoked the exception. We have implemented two different versions of this handler. The first one simply stalls the execution until the task become non-speculative, and then restarts the task. If the exception arises again, the application finishes with an error code. The second version implements "InclusiveSquash". In this case, this handler checks the state of the tasks as it was explained in Fig. 6. According with this checking, the handler may decide to squash all successors, returning the control within the parallel region for restarting the same task. Since each thread should be able to handle its own exceptions, it is necessary to indicate precisely what exceptions each thread will handle. To do so, when threads are spawned they initialize their own handler tables.

6.2. Benchmarks considered

To test both schemes, we have considered three applications that present a high number of dependence violations. These applications, widely used in Computational Geometry, are the construction of the two-dimensional Delaunay Triangulation (2D-DT), the construction of the two-dimensional

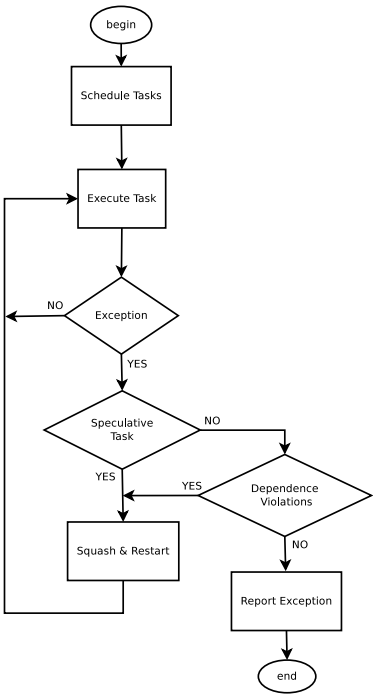

Figure 6. Flow diagram of sequence of actions taken when a exception arises.

Convex Hull (2D-Hull), and the construction of the two-dimensional Minimum Enclosing Circle (2D-MEC).

The first application is the randomized incremental construction of the Delaunay Triangulation using the *Jump-and-Walk* strategy, introduced by Mücke, Zhu *et al.* [12], [13]. This strategy proceeds in an incremental way. The Jump-and-Walk strategy uses a number of points, called anchors, whose containing triangles are known. The algorithm finds the closest anchor to the point to be inserted (the *jump* phase), and then traverses the current triangulation until the triangle that contains the point to be inserted is found (the *walk* phase). After this location step, the algorithm divides this triangle into three new triangles, and then updates the surrounding edges to keep the Delaunay properties. This local modification to the current Delaunay solution may lead to dependence violations, since other threads may have traversed the old solution while trying to add new points. The expected amount of dependence violations that may arise depends both on the number of processors and the length of the traversing path.

The 2D-Hull randomized incremental algorithm, due to Clarkson *et al.* [14], computes the convex hull (smallest enclosing polygon) of a set of points in the plane. The input to Clarkson's algorithm is a set of (x, y) point coordinates. The algorithm starts with the triangle composed by the first three points and adds points in an incremental way. If the point lies inside the current solution, it will be discarded. Otherwise, the new convex hull is computed, an operation that generates a dependence violation because other successor threads will be using the old solution. The probability of a dependence violation depends on the shape of the input set. If N points are distributed uniformly on a disk, the i-th iteration will present a dependence with probability in $\theta(\sqrt{i}/i)$. If points lie uniformly on a square, the probability of a dependence will be in $\theta(\log(i)/i)$. When a dependence is found, the convex hull has to be updated. The amount of work needed to do this is in $\theta(\log(i))$ in both cases.

Finally, the two-dimensional Minimum Enclosing Circle algorithm [15] computes the minimum enclosing circle of a two-dimensional, randomly-ordered point set. We have used a iterative version composed of three nested loops. Each loop fixes one of the two or three points that define the solution. We have speculatively parallelized the inner loop. This loop receives two candidates that define the limits of the enclosing circle and tries to find a third limit. Like the other benchmarks considered, this algorithm proceeds incrementally, discarding the third candidate if it discovers that a given point of the input set lies outside the proposed solution. This discard operation may to a dependence violation, since successors may have used the discarded points to test if other points lie inside the current solution.

6.3. Design of experimentation

The three benchmarks considered generate many dependence violations during their execution, posing a significant challenge to any STLS scheme. However, *none of the benchmarks considered generate any speculative exception during their execution*[1]. At this point we had two options: to develop a synthetic benchmark that artificially generates exceptions, or to augment the code of these well-known benchmarks to generate exceptions from time to time. We finally chose this second option, because to use real benchmarks avoids the need of choosing "adequate" workloads for a synthetic benchmarks in order to simulate real-world conditions.

To test both solutions to the speculative exceptions problem, we have slightly modified each benchmark, adding operations that lead to exceptions. We decided to develop two modified versions. One of them raises exactly one exception per each block of iterations being processed, while the second one raises two exceptions per block. Since the optimum block size is different for each application (20 iterations for 2D-DT, 2500 for 2D-HULL and 5000 for 2D-SEC), these choices in the design of experiments allow us to test both solution to the problems under very different execution conditions.

We have evaluated three aspects of parallel execution: The total wall-clock time needed for the speculative loop, the

1. Note that it is extremely unlikely that a widely-used benchmark allows the possibility of a runtime exception.

total number of dependence violations that arise, and the total number of squashes generated.

6.4. Experimental results

Figure 7 shows results for the Delaunay Triangulation, Fig. 8 shows the results for the Convex Hull problem, and Fig. 9 shows the results for the MEC problem. Results are consistent for all three applications, showing that it is preferable to handle exceptions immediately than to delay its processing until the non-speculative state reaches the thread that generates the exception. From these results we can draw the following observations:

- Due to the extraordinary heavy workload that represents the occurrence of an exception per each block of iterations being processed, execution time shown (leftmost plots) tend to increase with the number of processors. To mitigate this effect we might choose to raise a smaller number of exceptions, but in this case differences between both solutions are far less visible.
- The execution time is consistently smaller for the "InclusiveSquash" solution. In the case of the Delaunay Triangulation, our experimental results show a performance improving up to 13.44% when one exception per block arises, and up to 39.38% with two exceptions. The use of "InclusiveSquash" also lead to better results for 2D-HULL (43.77% for one exception and 41.09% for two), and for 2D-MEC (42.72% for one exception and 52.02% for two).
- As expected, the number of squashes generated by the "InclusiveSquash" solution is greater than those generated by the "SpinWait" solution (see center plots of Fig. 7, Fig. 8 and Fig. 9). The reason is that the experiments using the former solution have two source of squashes: Those generated by the normal execution of the application, and those generated by the speculative exceptions. Regarding the "SpinWait" solution, it does not generate other squashes than those produced by the original application's behavior. Interestingly, despite the higher number of squashes, the experiments that use the "InclusiveSquash" solution are consistently faster.
- Also as expected, the total number of exceptions risen is greater for the "InclusiveSquash" solution than those generated by the "SpinWait" solution (see rightmost plots of Fig. 7, Fig. 8 and Fig. 9). The reason is that the former solution executes several times the same block until the exception disappears.
- It is interesting to note the apparently strange behavior of the 2D-DT benchmark when running with 2 threads and one exception per block. The number of squashes (Fig. 7, first row, center) is extraordinary low (595 for "InclusiveSquash" and 591 for "SpinWait". This is about 1% of the squashes generated with four processors. The exceptions raised are also very low, being 118 and 117 respectively (Fig. 7, first row, right). Recall that, with two threads, one acts as the non-speculative and the other will be speculative. With the sliding window mechanism used by the STLS scheme used (see [4] and [8] for a detailed description), each time the non speculative thread finishes, it passes the "non speculative token" to the following thread. Since the workload generated by the Delaunay Triangulation algorithm is extremely regular, by the time the (initially) speculative thread generates the exception, its predecessor has already finished, and our thread has in fact become non speculative. Since only exceptions that are generated when the thread is still speculative count in our study, the exception is discarded, leading to the low number of exceptions shown and to an identical behavior for both solutions in terms of performance (Fig. 7, first row, left).

Finally, it is worthwhile to say that none of the studied solutions to the speculative exceptions problem lead to any performance degradation when exceptions are *not* produced. The only additional time needed for both solutions compared with a STLS system that does not use any of them is the time needed to register the handler, an operation that it is only carried out once per running thread.

6.5. Execution environment

Experiments were carried out on an Intel S7000FC4URE server, equipped with four quad-core Intel Xeon MPE7310 processors at 1.6GHz and 32GB of RAM. The system runs Ubuntu Linux operating system. All threads had exclusive access to the processors during the execution of the experiments, and we used wall-clock times in our measurements. The compilation flags for the sequential versions were `-O3 -m32 -march=native`. Times shown in the following sections represent the time spent in the execution of the main loop of the application. The time needed to read the input set and the time needed to output the results have not been taken into account.

7. Related work

Since parallel speculative execution exploits the use of data that could be incorrect to boost the speedup, polluted data can potential alter the original execution flow, thus producing unexpected results. These situations need to be kept under control to ensure a robust behavior. However, to the best of our knowledge little work has been carried out regarding robustness on software-based speculation so far. We will briefly mention the more relevant contributions.

Kulkarni et al. [5] focus their work on irregular applications written in object-oriented languages, exploiting speculative parallelism in objects instead of data. In their scheme, programmers are responsible of using programming

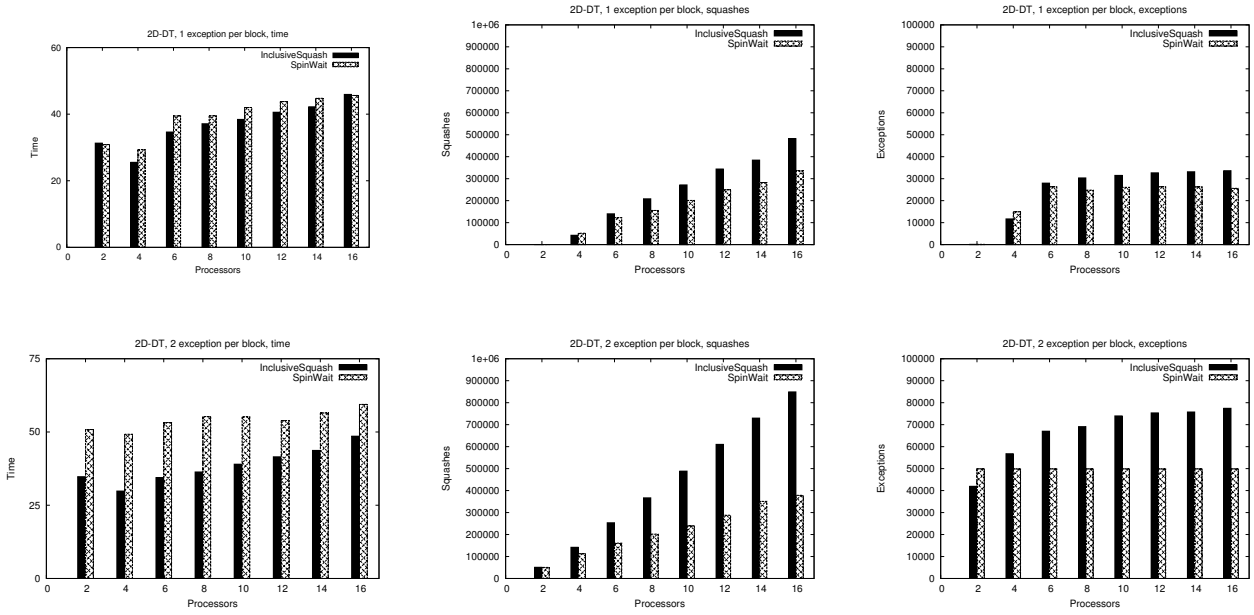

Figure 7. Experimental evaluation of speculative exception management using the Delaunay Triangulation.

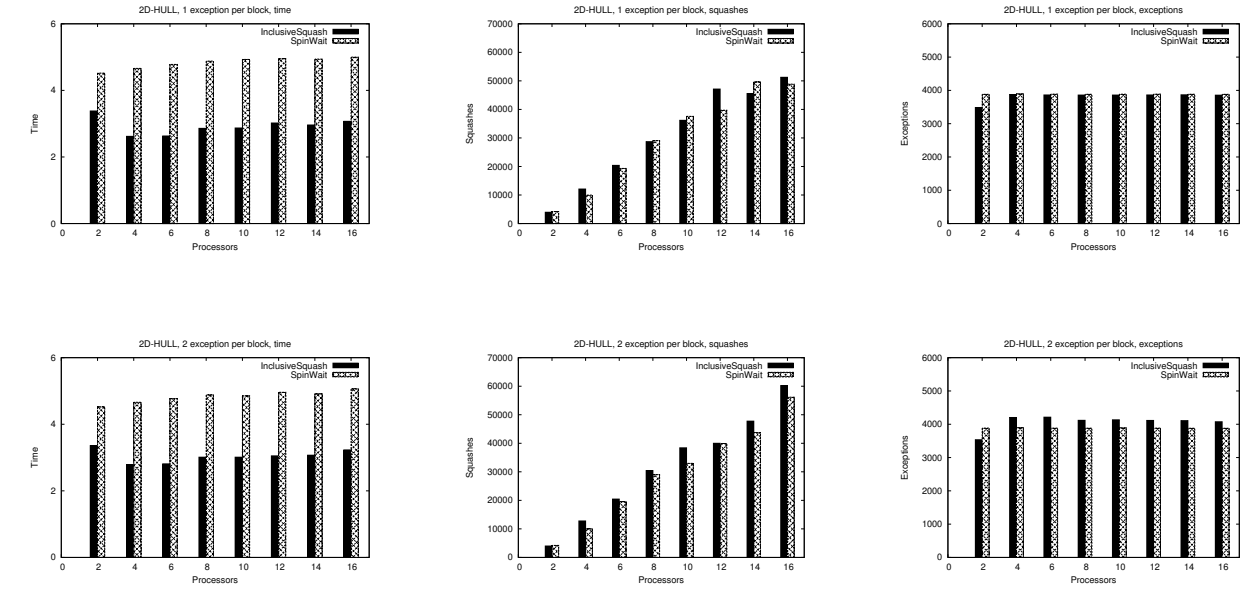

Figure 8. Experimental evaluation of speculative exception management using the 2-dimensional Convex Hull problem.

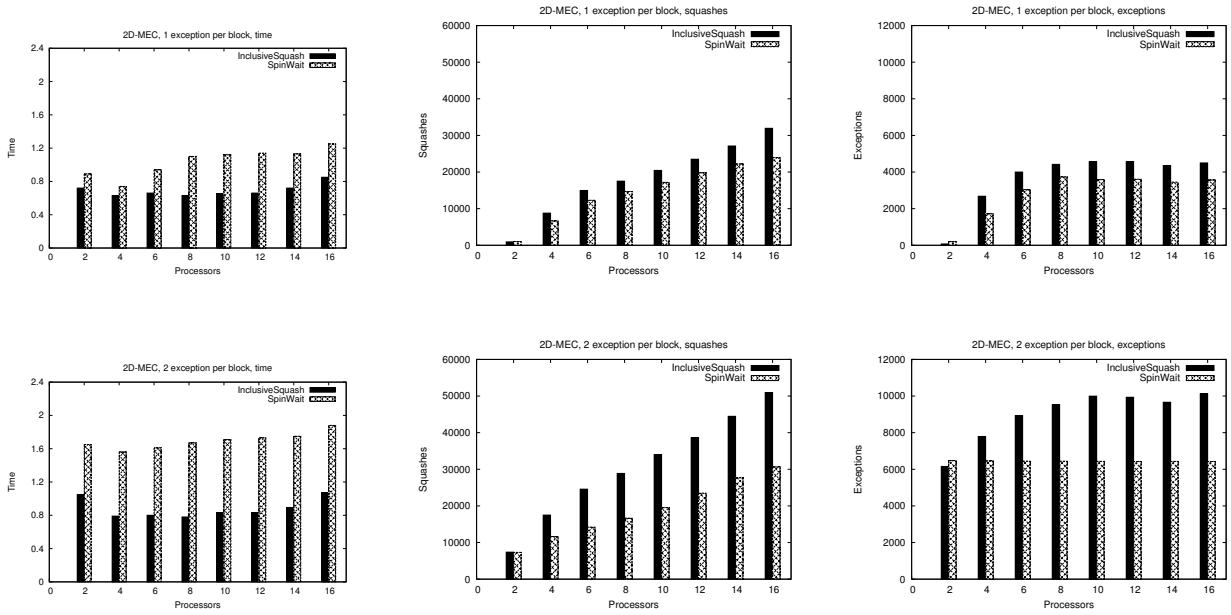

Figure 9. Experimental evaluation of speculative exception management using the 2-dimensional Minimum Enclosing Circle problem.

abstractions to highlight opportunities for exploiting parallelism in sequential programs, while a runtime system uses these hints to execute and monitorize the program in parallel. This mechanism needs to undo the results in shared objects when a dependence violations arises. No reference is given about robustness issues.

In [2], [16], a speculative software based on processes instead of threads is proposed. Programmers mark Potential Parallel Regions (PPR) that can be not only loop sections but also inherently sequential code. Several PPR are executed both sequentially and speculatively. At the end of a PPR execution, sequential and parallel results are compared in terms of memory pages and if they have made identical changes to the same memory locations speculative execution is considered to be correct. Otherwise the work by following PPRs is discarded. Since the proposed solution uses processes, memory pages are naturally protected to avoid segmentation faults. Moreover, as they not only execute the speculative code but also the sequential code, their scheme avoids in practice the problems mentioned in our work.

In [6], [17], a C++ library that contains different models of STLS is presented. Models from read, store and commit operations defers of each other. Using this scheme, dependency violations, task ends, and exceptions are handled in a similar way, with speculative threads waiting until becoming non speculative.

In [9], the proposed scheme captures all exceptions. A special handler activates a single flag that is used to denote all kinds of failure, both due to exceptions and dependence violations. A main thread is responsible of the management of all finished tasks in order. Therefore, the handling of any exception produced will be delayed until the associated task can be re-started as non speculative task. A previous work of the same group [3] pointed out that bzip2 application from SPEC2000 benchmark indeed produce exceptions when it was speculatively executed, but no further details are given. From our experience we deduce that the exceptions reported depend on the input set used to run the experiment.

In [18] STLS Java system is described. This system modifies bytecode instructions to provide speculation support. In this case, each time an exception arises the system returns to the statement that have produced the exception and the scheme waits until the main thread arrives to that execution point, in order to decide whether the data should be committed or discarded.

8. Conclusions

Robustness is a key issue for any software-based, thread-level speculation scheme that aims to be used in production runs. However, little work has been carried out to ensure that speculative, incorrect data do not alter irreversibly the sequential semantics of the original code in STLS systems. In this work we have identified the management of speculative exceptions as a key issue to ensure correct execution. We have proposed a new, eager mechanism that restarts the execution of speculative code as soon as an exception arises. We have compared our solution with a more conservative ap-

proach recently proposed, that waits until the thread becomes non speculative to solve the situation. Our experimental results on a real system with a state-of-the-art STLS scheme and three different benchmarks show that our solution leads to a performance improvement up to 52.02% in the total execution time. Our system has immediate applicability and, by its nature, does not lead to any performance degradation when speculative exceptions do not occur.

Acknowledgments

The authors would like to thank the anonymous referees for their detailed comments, and Álvaro Estébanez, David Orden, and Belén Palop for their help with the 2D-MEC benchmark and for many fruitful discussions. This research is partly supported by the Spanish Government (TIN2007-62302, TIN2011-25639, TIN2010-12011-E (CAPAP-H), CENIT MARTA, CENIT OASIS, CENIT OCEANLIDER). Part of this work was carried out under the HPC-EUROPA2 project (project number: 228398), with the support of the European Community - Research Infrastructure Action of the FP7.

References

[1] M. K. Prabhu and K. Olukotun, "Exposing speculative thread parallelism in SPEC2000," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '05. New York, NY, USA: ACM, 2005, p. 142–152, ACM ID: 1065964.

[2] K. Kelsey, T. Bai, C. Ding, and C. Zhang, "Fast track: A software system for speculative program optimization," in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '09. Washington, DC, USA: IEEE Computer Society, 2009, p. 157–168, ACM ID: 1545066. [Online]. Available: http://dx.doi.org/10.1109/CGO.2009.18

[3] C. Tian, M. Feng, V. Nagarajan, and R. Gupta, "Copy or discard execution model for speculative parallelization on multicores," in *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 41. Washington, DC, USA: IEEE Computer Society, 2008, p. 330–341, ACM ID: 1521785. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2008.4771802

[4] M. Cintra and D. R. Llanos, "Toward efficient and robust software speculative parallelization on multiprocessors," in *Proc. of the SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, June 2003, pp. 13–24.

[5] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," *Communications of the ACM*, vol. 52, p. 89–97, Sep. 2009, ACM ID: 1562188.

[6] C. E. Oancea, A. Mycroft, and T. Harris, "A lightweight in-place implementation for software thread-level speculation," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, ser. SPAA '09. New York, NY, USA: ACM, 2009, p. 223–232, ACM ID: 1584050.

[7] C. Tian, M. Feng, and R. Gupta, "Supporting speculative parallelization in the presence of dynamic data structures," in *ACM SIGPLAN Notices*, ser. PLDI '10. New York, NY, USA: ACM, 2010, p. 62–73, ACM ID: 1806604.

[8] M. Cintra and D. R. Llanos, "Design space exploration of a software speculative parallelization scheme," *IEEE Trans. on Paral. and Distr. Systems*, vol. 16, no. 6, pp. 562–576, June 2005.

[9] C. Tian, C. Lin, M. Feng, and R. Gupta, "Enhanced speculative parallelization via incremental recovery," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, p. 189–200, ACM ID: 1941580.

[10] F. Amerson, R. Gupta, V. Kathail, B. Rau, M. Schlansker, and W. Worley Jr, "Method and system for propagating exception status in data registers and for detecting exceptions from speculative operations with non-speculative operations," Jul. 7 1998, US Patent 5,778,219.

[11] "Openmp version 3.0 complete specifications," May 2008, http://www.openmp.org/mp-documents/spec30.pdf.

[12] L. Devroye, E. P. Mücke, and B. Zhu, "A note on point location in Delaunay triangulations of random points," *Algorithmica*, vol. 22, pp. 477–482, 1998.

[13] E. P. Mücke, I. Saias, and B. Zhu, "Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations," in *Proceedings of the 12th ACM Symposium on Computational Geometry*, 1996, pp. 274–283.

[14] K. L. Clarkson, K. Mehlhorn, and R. Seidel, "Four results on randomized incremental constructions," *Comput. Geom. Theory Appl.*, vol. 3, no. 4, pp. 185–212, 1993.

[15] E. Welzl, "Smallest enclosing disks (balls and ellipsoids)," in *New results and new trends in computer science*, vol. LNCS (555), 1991, pp. 359–370.

[16] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang, "Software behavior oriented parallelization," *ACM SIGPLAN Notices*, vol. 42, p. 223–234, Jun. 2007, ACM ID: 1250760.

[17] C. E. Oancea and A. Mycroft, "Software thread-level speculation: an optimistic library implementation," in *Proceedings of the 1st international workshop on Multicore software engineering*, ser. IWMSE '08. New York, NY, USA: ACM, 2008, p. 23–32, ACM ID: 1370090.

[18] C. J. F. Pickett and C. Verbrugge, "Software thread level speculation for the java language and virtual machine environment," *IN LCPC'05: Proceedings of the 18th intl. Workshop on Languages and compilers for parallel computing*, vol. LNCS 4339, pp. 304—318, 2005.