Just-In-Time Scheduling for Loop-based Speculative Parallelization

Diego R. Llanos
Departamento de Informática
Universidad de Valladolid
47011 Valladolid, Spain
diego@infor.uva.es

David Orden
Departamento de Matemáticas
Universidad de Alcalá
Alcalá de Henares, Madrid, Spain
david.orden@uah.es

Belén Palop
Departamento de Informática
Universidad de Valladolid
47011 Valladolid, Spain
b.palop@infor.uva.es

Abstract

Scheduling for speculative parallelization is a problem that remained unsolved despite its importance. Simple methods such as Fixed-Size Chunking (FSC) need several 'dry-runs' before an acceptable chunk size is found. Other traditional scheduling methods were originally designed for loops with no dependences, so they are primarily focused in the problem of load balancing. In general, all these methods perform poorly when used for speculative parallelization, where loops may present unexpected dependences that adversely affect performance.

In this work we address the problem of scheduling loops with and without dependences for speculative execution. We have found that a trade-off between minimizing the number of re-executions and reducing overheads can be found if the size of the scheduled block of iterations is calculated at runtime. We introduce here a scheduling method called Just-In-Time (JIT) scheduling that uses the information available during the execution of the loop in order to dynamically compute the size of the next block to be scheduled. The results show a 10% to 26% speedup improvement in real applications with dependences with respect to a carefully-tuned FSC strategy, and a 9% to 39% speedup improvement in real applications without dependences. With our proposal, the number of dependence violations that lead to squashes can be reduced by up to 62%. Moreover, in applications where the cost of dependence violations is too high to obtain speedups with FSC, our runtime scheduling mechanism avoids performance degradation.

Keywords: *Speculative parallelization, loop-based speculation, speculative multithreading, scheduling.*

1. Introduction

Speculative parallelization (also called *thread-level speculation*) is the most promising technique for automatic extraction of parallelism of irregular loops. With speculative parallelization, loops that can not be analyzed at compile time are optimistically executed in parallel. Hardware or software mechanisms ensure that all threads access the shared data according to sequential semantics. A *dependence violation* appears when one thread incorrectly consumes a datum that has not been generated by a predecessor yet. In the presence of such a violation, earlier software-only speculative solutions (see, e.g. [12, 14]) interrupt the speculative execution and re-execute the loop serially. More recent approaches [4, 6, 15] squash only the offender thread and its successors, re-starting them with the correct data values.

It is easy to understand that frequent squashes adversely affect speculation performance. One way to reduce the cost of a squash is to assign smaller subsets (called *chunks*) of iterations to each thread, reducing the amount of work being discarded in the case of a squash. A correct choice of the chunk sizes is critical for speculation performance. However, smaller chunks also imply more frequent commit operations and a higher scheduling overhead. Most of the scheduling methods proposed so far in the literature deal with independent iterations, and do not take into account the cost of re-executing threads.

In this work we address the problem of scheduling for speculative execution of loops with and without dependences. We have found that a trade-off between minimizing the number of re-executions and reducing overheads can be found if the size of the scheduled block of iterations is calculated at runtime. We introduce here a scheduling method called Just-In-Time (JIT) scheduling. This method uses the information available during the execution of the loop in order to dynamically compute the size of the next block to be scheduled.

The rest of the paper is organized as follows. Section 2 reviews the existent scheduling alternatives currently used with speculative parallelization. Section 3 shows that the information available at run time allows to issue chunks of an appropriate size, and introduces JIT scheduling. Section 4 explores two different design options for JIT scheduling,

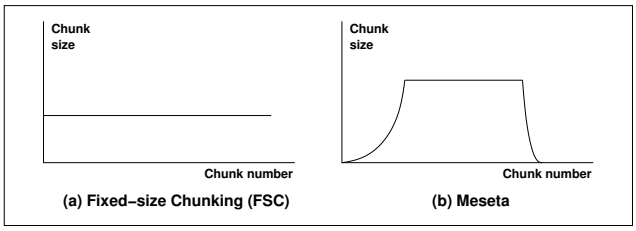

Figure 1. Scheduling methods used so far with speculative parallelization.

analyzing their advantages and drawbacks. Section 5 shows an experimental evaluation of our scheduling mechanism in the execution of real applications with and without dependences. Finally, Section 6 concludes the paper.

2. Review of scheduling alternatives for speculative execution

The problem of scheduling independent iterations of loops among different processors in a parallel system has been extensively studied in the literature. Most of the proposed scheduling methods were designed in a context of independent iterations, and their main goal was to balance the workload among processors (see [7, 8, 9, 13, 17, 18]). To do so, most of the functions that compute chunk sizes issue chunks of decreasing size. However, these mechanisms perform poorly in the context of speculative parallelization, where loops may present dependences among iterations and the costs associated to them are higher than those due to load imbalance [11].

To the best of our knowledge, only two scheduling methods have been used until now in real executions of loops that may present dependences among iterations: Fixed Size Chunking (FSC) and MESETA.

The method called *Fixed Size Chunking* (FSC), proposed by Kruskal and Weiss [9], consists of choosing a chunk size and assigning a block of iterations with that size to each processor. The efficiency of this scheme heavily depends on the choice of an appropriate value for the chunk size, a difficult task for both programmers and compilers. Kruskal and Weiss give the following formula for the optimal value of the chunk size, K_{opt}:

$$K_{\text{opt}} = \left(\frac{\sqrt{2} N h}{\sigma P \sqrt{\log P}} \right)^{2/3},$$

where σ is the variance of the iteration time, h the scheduling overhead, N the number of iterations and P the number of processors. The first three values are unknown at the beginning of the loop, making it difficult to determine the optimal (or at least adequate) chunk size in practice. Therefore, the only way to accurately choose the chunk size for a particular application, even with no dependences, is by experimentation. Despite these limitations, its simplicity made FSC the default choice for speculative parallelization. Figure 1(a) shows the shape of the FSC function.

In speculative parallelization loops may present dependences among iterations, making even more difficult to find a good scheduling mechanism. A *Read-after-write* (RAW) dependence violation appears when a thread speculatively reads a value and, later, a predecessor modifies the same value. If a dependence violation occurs during the parallel execution of the loop, the offending thread and all of its sucessors are squashed and restarted in order to consume the correct values.

It is difficult to choose an "optimal", fixed chunk size in this context. In general, big chunk sizes increase the time spent on re-executions, while smaller chunks increase the overhead time [11]. Therefore, a good strategy for speculative parallelization is to issue big chunks of iterations on portions of the loop where fewer dependences are expected, and smaller chunks on portions where we expect to find many dependences. The problem here is how to determine *where* more dependences will appear, since the particular dependency pattern for a loop depends on both the algorithm and its input set. This makes a given chunk size acceptable for a particular combination of both but suboptimal for a different one.

MESETA [10, 11] is a scheduling mechanism designed for the speculative execution of randomized incremental algorithms, where the dependence profile is known beforehand. MESETA assigns chunks with different sizes as the execution proceeds. The behavior of MESETA (Spanish word for tableland) divides the parallel execution in three stages. Small chunks of growing size are scheduled at the beginning, since most of the dependences that appear in randomized incremental algorithms happen at the first iterations of the loop. At a certain point, growth stops and MESETA behaves like FSC. Finally, chunk sizes decrease in order to balance the load among processors. Figure 1(b) shows the shape of the MESETA function.

MESETA takes advantage of the fact that most dependence violations in randomized incremental algorithms occur while executing the first iterations. However, this solution is not valid for applications where the dependence pattern is not known beforehand. Besides this, the fixed chunk size for the stable part of MESETA should also be determined by experimentation, and it depends not only on the particular algorithm being parallelized, but also on the input set considered.

Despite its importance, the general problem of scheduling for speculative parallelization remained unsolved.

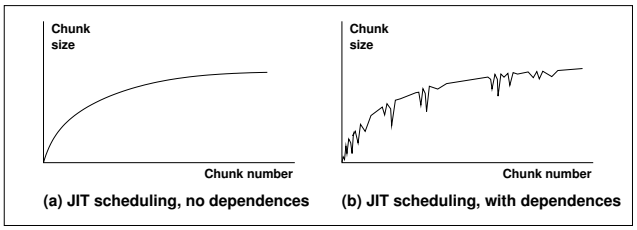

Figure 2. Runtime scheduling in loops with and without dependences.

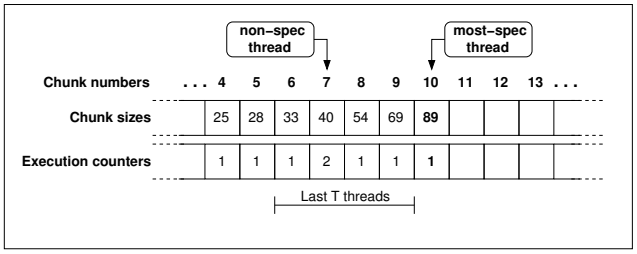

Figure 3. JIT scheduling, conservative approach. The first time chunk 10 is executed, its size is calculated using the JIT scheduling function and its execution counter is set to 1. The size will be preserved regardless of the number of re-executions of chunk 10.

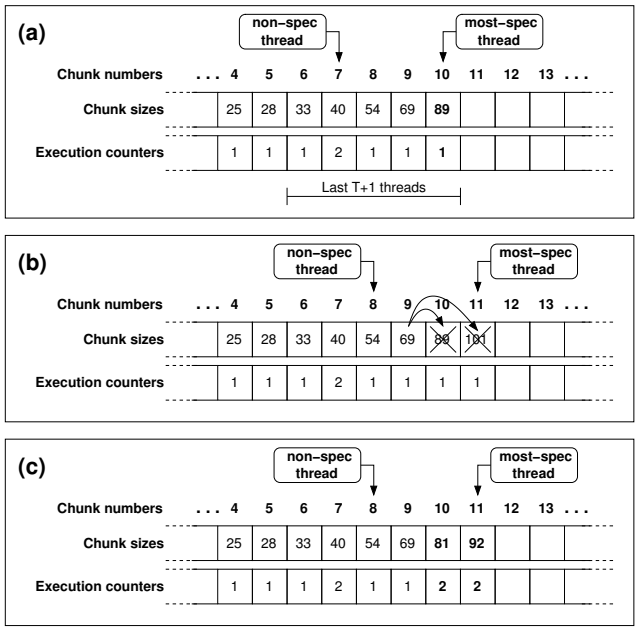

Figure 4. JIT scheduling, aggressive approach. (a) Size of chunk 10 is calculated using the JIT scheduling function. (b) Chunk 9 issues a squash operation. (c) Squashed threads recalculate in program order the size of the chunk to be executed, using the new value of the execution counters.

3. Just-In-Time scheduling

As we saw in the previous section, no known scheduling method tries to solve the general problem of scheduling loops with dependences. Either they do not take into account dependences between iterations, have to be carefully tuned through experimentation, and/or fit a particular type of algorithms. In this section we propose a new scheduling mechanism that deals with this situation.

In order to decide the size for the next chunk of iterations to be scheduled, let $e(t)$ be the total number of times the previous t chunks were executed, including the number of re-executions due to mispeculation (for simplicity, we consider all iterations of equal size). If no dependence violations among threads have arisen, $e(t) = t$, since each chunk has been executed exactly once. Using as a general guidance the average number of executions per chunk $\overline{e}(t) = \frac{e(t)}{t}$, a scheduling mechanism may issue larger chunks if no dependence violations have arisen lately, while issuing smaller chunks if the last chunks had to be squashed and re-executed several times.

The number of re-executions needed for each chunk is not known at compile time, but can be easily obtained at runtime, just by maintaining a set of execution counters, one for each chunk scheduled. A thread that starts the execution of the following chunk can use $\overline{e}(t)$ to compute its own chunk size.

The runtime value of the execution counters is not the only source of information that can be used to calculate the following chunk size. An important datum that should be taken into account is the total number of iterations, N. This value allows to compute a chunk size dependent on the size of the entire loop, since issuing chunks of one thousand iterations may be acceptable for a loop composed by one million iterations, but will lead to a severe load imbalance if the loop has only, for example, $5\,234$ iterations and we are dedicating eight processors to this task. In a similar way, is important to take into account the index i of the first iteration of the chunk to be scheduled. Since the dependency pattern of the loop being scheduled is unknown, it is reasonable to start with small chunks and adjust their size using the runtime information as execution proceeds. It is ongoing work in this topic to obtain a JIT mechanism with a smaller dependence on i and with a similar performance gain as the one described here.

The scheduling method proposed in this paper takes into account all three values $\overline{e}(t), N$ and i. The thread commissioned to execute the following chunk will calculate the chunk size C taking into account these values, making the

chunk size dependent on N and i and inversely proportional to $\overline{e}(t)$. We propose in this paper a general scheduling function based in these values to calculate the chunk size. The formula is

$$C = \left\lceil \frac{\ln(i) \cdot \ln(N)}{\overline{e}(t)} \right\rceil \quad (1)$$

If no dependences have arisen during the execution of the last t chunks, the last t execution counters will be one, and $\overline{e}(t)$ will also be one. In this case, the function will have the shape depicted in Fig. 2(a). On the other hand, if some of the last t chunks have been squashed, their execution counters will be greater than one. This will make $\overline{e}(t)$ also greater than one, and the size of the following chunk size will not be as big as if no dependences have arisen. After some chunks the situation will hopefully be stable again, and consequently the scheduling function will present some "potholes", as shown in Fig. 2(b). The value of t can be fixed or can change with the number of processors. We propose a value for t proportional to the number of processors in the system. Finally, the logarithmic function smooths the increasement in the chunk size as N and i grow.

Note that, if no dependences arise, Eq. 1 leads to bigger chunk sizes as execution proceeds. This fact conflicts with the desire of obtaining a good load balancing towards the end of the loop, but, as we said in Sect. 2, to avoid the performance degradation due to mispeculation is more important for speculation performance.

The function given by Eq. 1 delivers small chunk sizes, a situation desirable if the loop has up to a few thousands of iterations. For example, in a loop composed by $3\,000$ iterations and with no dependences, the maximum chunk size attainable at the end of the loop (say, at iteration $2\,900$) is $C = \lceil \ln(2\,900).\ln(3\,000) \rceil = 64$ iterations. If the loop has one million iterations, the maximum value of C will be just 190 iterations. Delivering such small chunks in loops with many iterations increases the execution overheads. Therefore, for such loops it is desirable to let C grow faster:

$$C = \left\lceil \frac{\ln(i)^2 \cdot \ln(N)}{\overline{e}(t)} \right\rceil \quad (2)$$

As an example, the maximum value for C returned by Eq. 2 in a loop with one million iterations is 2637 iterations, thus reducing the execution overheads. As we will see in our experiments, we use bigger powers of $\ln(i)$ to produce bigger chunk sizes, while preserving the general philosophy of the JIT scheduling function.

Finally, it is interesting to note that any base for the logarithmic function used in Eqs. 1 and 2 will lead to a similar behavior than the use of $\ln(x)$. In our experiments we have used $\ln(x)$ because the cost of its calculation was small enough, representing less than 0.01% of the experiments' execution time. This time might be reduced further using base-2 logarithms.

4. Design space of Just-In-Time scheduling

The scheduling function proposed above lets each thread calculate its chunk size just before starting the execution of that chunk. At this moment, the thread that starts the execution of a new chunk will be the most speculative, and it will be able to read the execution counters of all its predecessors. The most-speculative thread will then calculate the chunk size, set to one its own execution counter, and begin the execution of its chunk of iterations, starting with the last iteration of the preceding chunk plus one. To avoid race conditions with other threads, the reads and updates of the execution counters are performed inside a critical section. Figure 3 shows the process in more detail.

But what happened if our thread is squashed? This means that a predecessor has detected a dependence violation and issued a squash event to all its successors. In this case we have two options. The first one is to keep the chunk size already calculated for this thread. Since the chunk size calculations are carried out inside a critical section, to keep the value already calculated has the advantage of reducing execution overheads. Note that this solution makes unnecessary to take into account the value of the execution counter of our thread in the calculations, because each chunk size will be calculated only once, and at that moment the value of our execution counter will always be one, thus not giving any useful information.

This behavior, while simple, has the drawback of not taking into account the change of the situation due to an eventual squash operation. If a given thread issues a squash operation, only threads that start for the first time the execution of a new chunk ahead of the most-speculative chunk squashed will use the updated values of the execution counters. Squashed threads, however, will remain unnoticed.

Therefore, it makes sense to recalculate the chunk size of all squashed threads to take into account the changes in the execution counters. Figure 4 shows this behavior. Note that now we examine $t + 1$ execution counters, since the execution counter associated to the current chunk should be also taken into account. As we will see in the following section, this second option delivers the best performance, while incurring in negligible overheads.

5. Experimental results

A state-of-the-art, software-only speculative parallelization engine [3, 4] was used to execute in parallel five different applications that present non-analyzable loops with and without dependences among iterations.

Algorithm	Input set description	Loop parallelized	Loop time as % of total time	Spec data size in KB	Iterations per invocation	% of dependence violations	Cost of squashing iter. i	FSC chunk size used (iterations)
TREE	Off-axis parab. collision	accel_10	94	< 0.1	4 096	0	n/a	2
MDG	Reference input set	interf_1000'	86	13	343	0	n/a	2
WUPWISE	Reference input set	muldeo_200', muldoe_200'	41	12 000	8 000	0	n/a	2
2D-Hull	Square, 10M points	Main loop	99	13	9 999 997	2.00	$O(\log i)$	5 000
2D-Hull	Disc, 10M points	Main loop	99	86	9 999 997	15.48	$O(\sqrt[3]{i})$	2 500
2D-SEC	Disc, 10M points	Main loop	99	< 0.1	10 000 000	< 0.01	$O(i)$	1 000

Table 1. Characteristics of the algorithms and input sizes used.

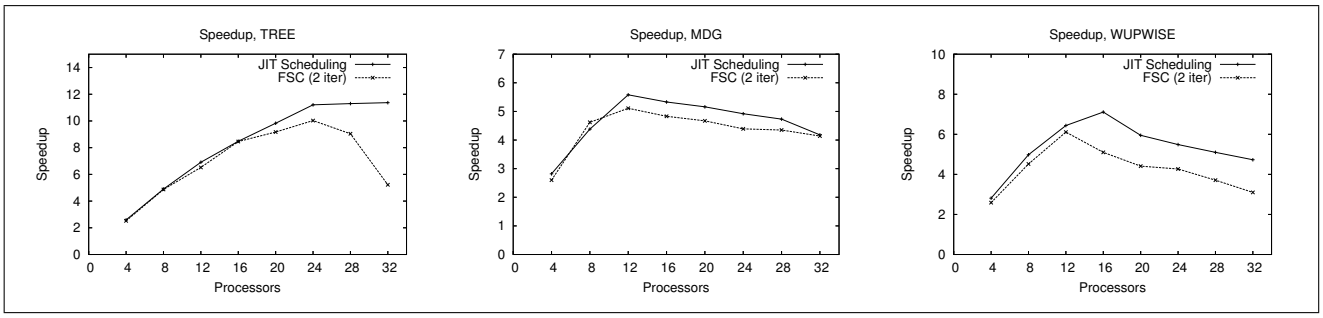

Figure 5. Speedups obtained in the execution of different applications with no dependences.

5.1. Applications considered

We consider in total five different applications. The first three applications have non-analyzable loops that do not suffer from dependence violations. The applications chosen were TREE from [1], MDG from the PERFECT Club benchmark suite [2] and WUPWISE from SPECfp2000 [16]. These applications are representative of legacy as well as recent sequential scientific programs. They spend a large fraction of their sequential execution time on loops that can not be automatically parallelized by state-of-the-art compilers because they have dependence structures that are either too complicated to be analyzed at compile time or dependent on the input data.

We consider two additional applications that present loops with dependences. Both applications implement solutions to well-known geometrical problems. As we will see, changes in the size or shape of the input set will allow us to change the expected number of dependences.

The first application with dependences, called 2-Dimensional Convex Hull (2D-Hull), is due to Clarkson *et al.* [5]. The algorithm computes the convex hull (smallest enclosing polygon) of a set of points in the plane. The convex hull is widely used in Computer Graphics, Motion Planning, Robotics or Computed Vision. The input of Clarkson's algorithm is a set of x-y point coordinates. The algorithm proceeds in an incremental way, computing at iteration i the convex hull of the first i points in the set. This way, if N points are distributed uniformly on a disk, the i-th iteration presents a dependency with probability in $O(\sqrt{i}/i)$. On the other hand, if points lie uniformly on a square, the probability of a dependence violation is in $O(\log(i)/i)$. When a dependence is found, the convex hull has to be updated. The amount of work needed to do this is in $O(\log i)$ in both cases. We generated two input sets of 10 million points (on a disk and a square). We have shuffled the input sets to guarantee the randomized order in which they are processed.

The second application, called 2-Dimensional Smallest Enclosing Circle (2D-SEC), is due to Welzl [19]. The algorithm finds the smallest enclosing circle containing a given set of points in the plane. This algorithm has been deeply studied in areas such as Computational Geometry, Optimization and Operations Research. The input of this algorithm is also a set of x-y point coordinates, while the output is a set of points (two or three, depending on their relative position) anchoring the circle, together with the center and radius of the smallest enclosing circle. The construction is also incremental and the ball enclosing the first i points is found at the i-th iteration. In this case, a dependence violation forces not only an update of the actual solution, but the recalculation of the entire enclosing ball. This computation takes linear time in the number of points already included. As we will see, this fact produces devastating effects in the performance of the speculative parallelization version.

Table 1 summarizes the characteristics of each application considered.

5.2. Environment setup

The experiments were performed on a Sun Fire 15K symmetric multiprocessor (SMP), equipped with 900MHz UltraSparc-III processors, each one with a private 64 KByte, 4-way set-associative L1 cache, a private 8 MByte, direct-mapped L2 cache, and 1 GByte of shared memory per processor. The system runs SunOS 5.9. The application was compiled with the Forte Developer 7 Fortran 95 compiler using the highest optimization settings for our execution environment: `-O3 -xchip=ultra3 -xarch=v8plusb -cache=64/32/4:8192/64/1`. 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. The application had exclusive use of the processors during the entire execution and we use wall-clock time in our measurements.

5.3. Performance of applications with no dependences

Figure 5 shows the relative performance of TREE, MDG and WUPWISE loops when executed with the same software-based speculation engine [4] and two different scheduling mechanisms: JIT scheduling using Eq. 1 and Fixed-Size Chunking (FSC). No special tuning of JIT was needed, while for FSC we had to determine a good chunk size by experimentation, trying with several values and running them with different numbers of processors. In Eq. 1 we have used a value for t equal to two times the number of processors.

The results show a speedup gain of 17.82% for TREE, a gain of 9.20% for MDG and a gain of 39.41% for WUPWISE. Note that JIT scheduling also avoids the performance degradation due to an increment in the number of processors: the most relevant example is the 117.82% gain for TREE when comparing the 32-processors performance with respect to FSC.

For these applications, both the conservative and the aggressive JIT mechanisms discussed in Sect. 4 are equivalent, since no squashes are issued and therefore the chunk size for these applications is calculated only once for each chunk.

5.4. Performance of the 2D Convex Hull

As we saw in Sect. 5.1, we have used two different input sets with the 2D Convex Hull algorithm in order to test the sensitivity of our proposal with respect to the number of dependences. The two plots on the left side of Fig. 6 show the speedups obtained while processing the disc-shaped and square-shaped input sets of 2D-Hull using three different scheduling alternatives. The mechanisms implemented were conservative JIT scheduling (where chunk sizes are calculated only once), aggressive JIT scheduling (where chunk sizes are re-calculated each time a thread is squashed), and the classic FSC mechanism.

As expected, the higher the number of dependences, the better the aggressive JIT scheduling performs with respect to its conservative counterpart. Results for the disc-shaped input set show a 26.00% performance improvement of the aggressive JIT scheduling with respect to FSC, and a 15.50% for the more conservative implementation. Regarding the square-shaped input set, where less dependences appear, results show a 15.98% performance improvement of the aggressive JIT scheduling with respect to FSC, and a 10.37% for the more conservative implementation.

The right column of Fig. 6 helps to understand the real effect of reducing at runtime the block size. The two plots show the number of dependence violations among different threads issued during the execution of the experiments. As can be seen, FSC raises much more dependences than our JIT scheduling mechanisms, due to the fact that FSC chunks are fixed-sized and many of them need to be recomputed several times until results follow sequential semantics. JIT mechanisms perform better, reducing the number of dependence violations that lead to squashes by up to 62%. The reason is that to issue smaller chunks in a context rich in dependence violations facilitates the commitment of partial results and avoids the production of further squashes.

To complete our analysis, Fig. 7 shows an execution time breakdown comparison for the three scheduling mechanisms considered. For this figure we can draw the following observations:

- The "busy" time, representing the time spent computing instructions present in the original application, is in general bigger for FSC, since the higher number of re-executions affect the time spent in these instructions.
- The "contention" time, representing the time the threads wait for access to the critical section used for scheduling threads, is much bigger for the disc input set, since squashes occur more frequently. As expected, contention is slightly higher for the aggressive JIT mechanism, due to the recalculation of the chunk size each time a thread is squashed. Despite this bigger contention time, overall speedups are better for the more aggressive JIT mechanism.

5.5. Performance of the 2D Smallest Enclosing Circle

As we stated in Sect. 5.1, Welzl's Smallest Enclosing Circle algorithm presents a comparatively high cost for

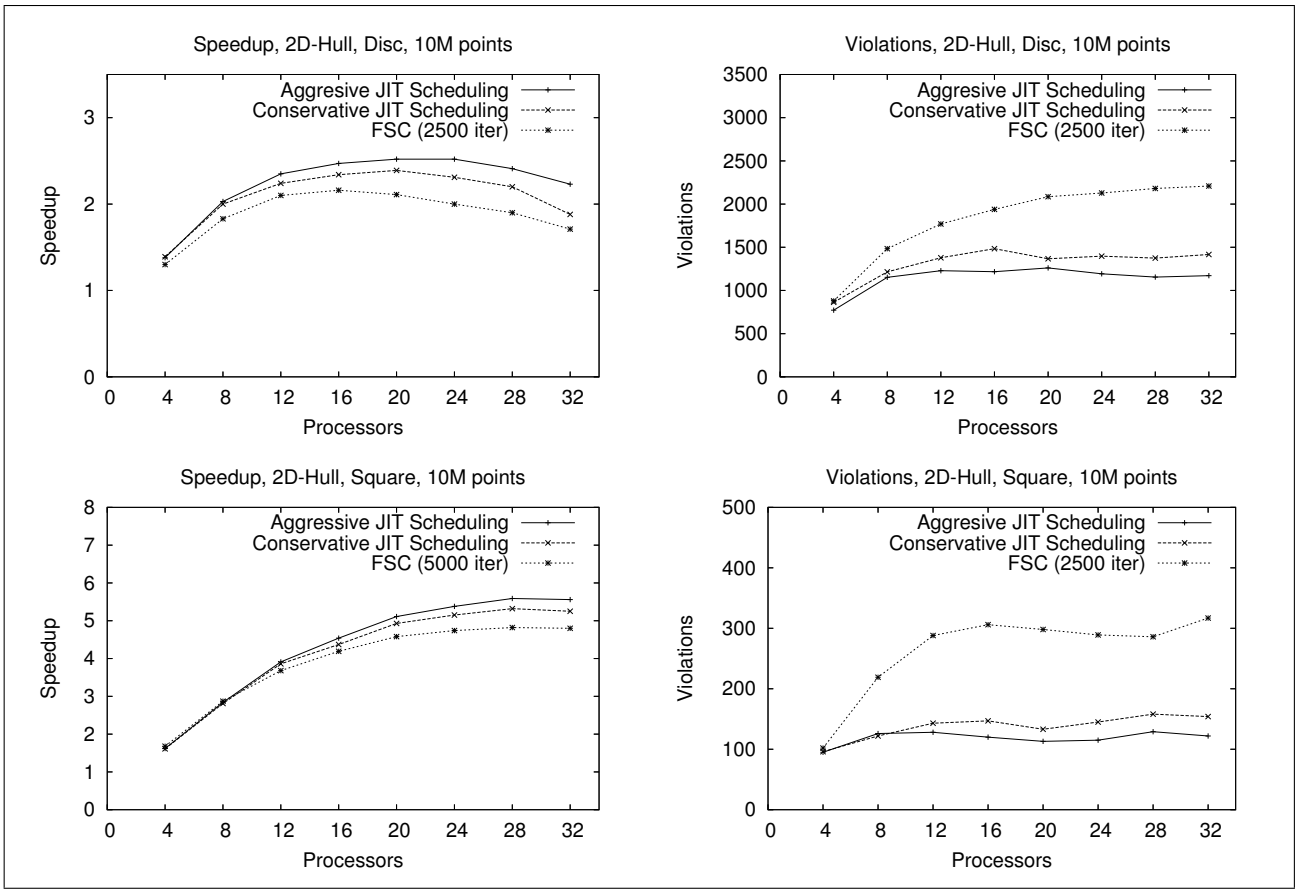

Figure 6. Speedups and number of dependence violations during the execution of the Randomized Incremental 2D-Hull algorithm.

solving each dependence violation. Each time a point is located outside the enclosing circle, all preceding points should be re-processed in order to obtain the new solution. This makes the cost of a dependence violation to be in $O(i)$, serializing in practice any attempt to parallelize the application, even by hand.

Figure 8 shows the speedups and number of dependence violations that arise while working with this algorithm. Welzl's algorithm runs in expected linear time, and since the cost of a dependence violation is also linear, it is not surprising that the maximum speedup obtained is 1.09. As expected, increasing the number of processors has a very negative impact on FSC performance, since the overhead grows with the number of processors but we are still unable to extract any parallelism of the application. However, both versions of the JIT scheduling mechanism curve the performance degradation. The reason is that the high number of re-executions makes JIT issuing very small blocks of iterations, thus performing frequent commits of processed chunks and letting the parallel execution progress.

It is interesting to note that both versions of JIT lead to *more* dependence violations than FSC for this particular algorithm. However, the overall total number of violations is quite small, and the more efficient scheduling generated by JIT makes this effect negligible.

6. Conclusions

In this work we address the problem of scheduling for speculative execution of loops with and without dependences. We have found that a trade-off between minimizing the number of re-executions and reducing overheads can be found if the size of the scheduled block of iterations is calculated at runtime. We have introduced a scheduling method called Just-In-Time (JIT) scheduling that uses the information available during the execution of the loop in order to dynamically compute the size of the next block to be scheduled. We explored the design space of this solution, from conservative positions like maintaining the block size chosen even if a dependence violation appears,

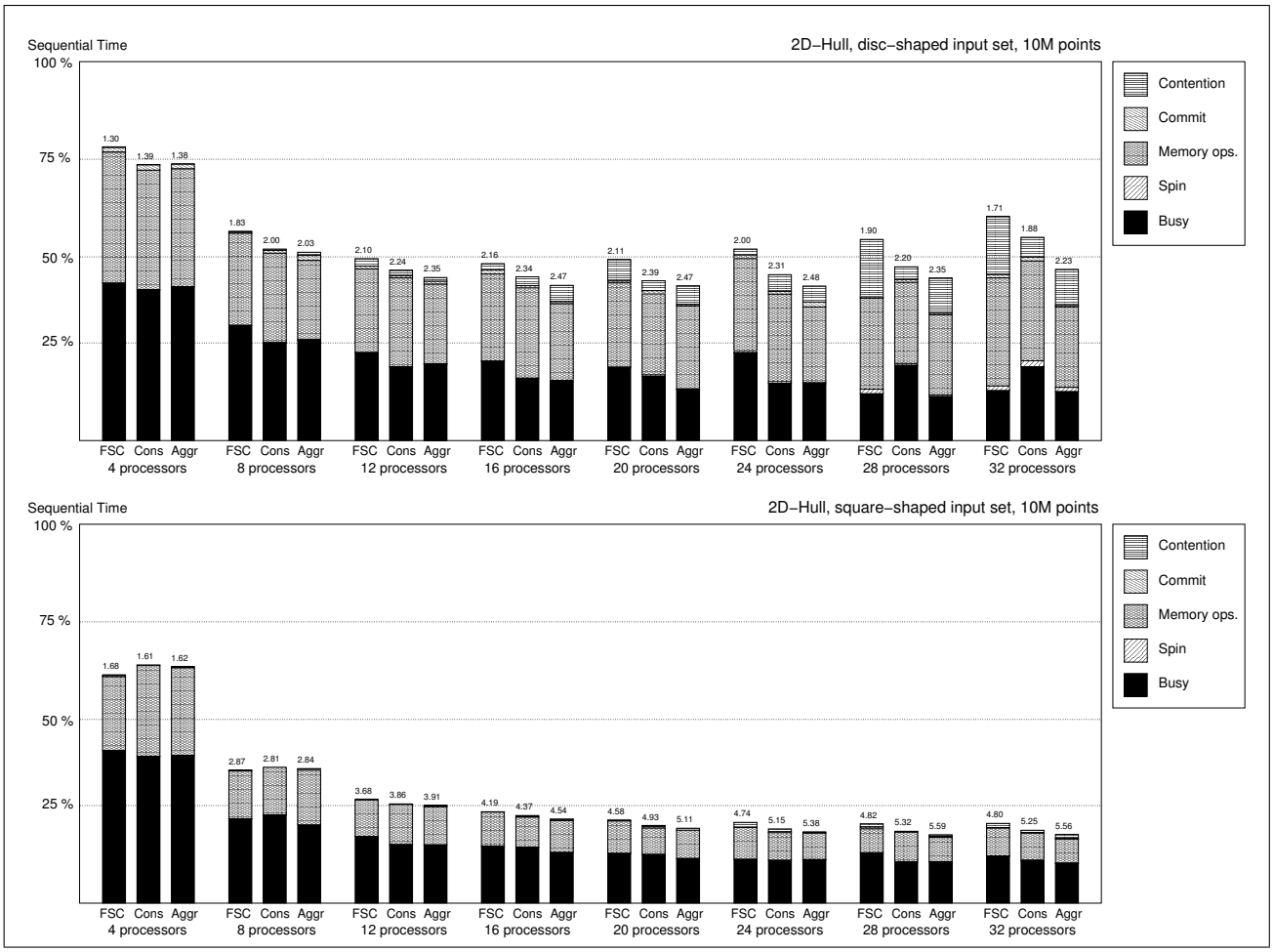

Figure 7. Execution time breakdowns for the 2-Dimensional Convex Hull.

to more aggressive choices like recalculating the best block size each time the thread is squashed. Results show a 10% to 26% speedup improvement in real applications with dependences with respect to a carefully-tuned FSC strategy, and a 9% to 39% speedup improvement in real applications without dependences, while the number of dependence violations that lead to squashes are reduced by up to 62%. We have also shown that JIT scheduling helps to avoid the performance degradation in applications where the cost of dependence violations is too high to obtain speedups.

Acknowledgments

The authors would like to thank the anonymous referees for their valuable suggestions. This work is being supported by grants VA031B06 (Junta de Castilla y León, Spain) and TIN2007-62302 (Ministerio de Educación y Ciencia, Spain). David Orden is also supported in part by grants MTM2005-08618-C02-02 and S-0505/DPI/0235-02. The authors would also like to thank the EPCC (Edinburgh Parallel Computing Center) for the main computing resources used in this work and its support staff, in particular, Chris Johnson and Catherine Inglis.

References

[1] J. E. Barnes. Institute for Astronomy, University of Hawaii, ftp://ftp.ifa.hawaii.edu/pub/barnes/treecode/.

[2] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Scheider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, 1989.

[3] M. Cintra and D. R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *Proceedings of the SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, June 2003.

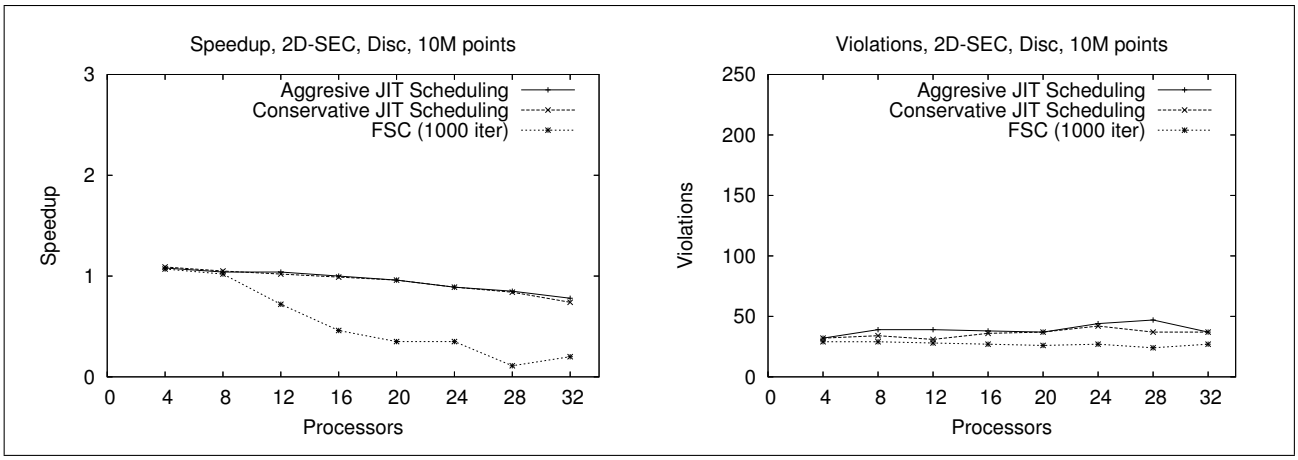

Figure 8. Speedups and number of dependence violations during the execution of the Randomized Incremental 2D-Smallest Enclosing Circle algorithm.

[4] M. Cintra and D. R. Llanos. Design space exploration of a software speculative parallelization scheme. *IEEE Trans. on Paral. and Distr. Systems*, 16(6):562–576, June 2005.

[5] K. L. Clarkson, K. Mehlhorn, and R. Seidel. Four results on randomized incremental constructions. *Comput. Geom. Theory Appl.*, 3(4):185–212, 1993.

[6] F. Dang, H. Yu, and L. Rauchwerger. The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In *Proc. of the 16th International Parallel and Distributed Processing Symposium (IPDPS '02)*, April 2002.

[7] T. Hagerup. Allocating independent tasks to parallel processors: An experimental study. *J. Parallel Distrib. Comput.*, 47(2):185–197, 1997.

[8] S. F. Hummel, E. Schonberg, and L. E. Flynn. Factoring: A Method for Scheduling Parallel Loops. *Communications of the ACM*, 35(2):90–100, August 1992.

[9] C. P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, SE-11(10):1001–1016, 1990.

[10] D. R. Llanos, D. Orden, and B. Palop. Meseta: A new scheduling strategy for speculative parallelization of randomized incremental algorithms. In *Proc. 2005 ICPP Workshops (HPSEC-05)*, pages 121–128, Oslo, Norway, June 2005. ISBN 0-7695-2381-1, IEEE Press.

[11] D. R. Llanos, D. Orden, and B. Palop. New scheduling strategies for randomized incremental algorithms in the context of speculative parallelization. *IEEE Transactions on Computers*, 56(6):839–852, 2007.

[12] M. Gupta and R. Nim. Techniques for run-time parallelization of loops. *Supercomputing*, November 1998.

[13] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, December 1987.

[14] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Conf. on Programming Languages Design and Implementation*, pages 218–232, June 1995.

[15] P. Rundberg and P. Stenström. Low-Cost Thread-Level Data Dependence Speculation on Multiprocessors. In *Workshop on Scalable Shared Memory Multiprocessors*, June 2000.

[16] Standard Performance Evaluation Corporation. http://www.spec.org/.

[17] P. Tang and P.-C. Yew. Processor self-scheduling for multiple nested parallel loops. In *IEEE Intl. Conf. on Parallel Processing*, pages 528–535, August 1986.

[18] T. H. Tzen and L. M. Ni. Trapezoid self-scheduling: A pratical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, 1993.

[19] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *New Results and new Trends in Computer Science*, number 555 in Lecture Notes in Computer Science, pages 359–370. Springer, 1991.