Moody Scheduling for Speculative Parallelization

Alvaro Estebanez[1], Diego R. Llanos[1](✉), David Orden[2], and Belen Palop[1]

[1] Dpto. Informática, Universidad de Valladolid,
Campus Miguel Delibes, 47011 Valladolid, Spain
{alvaro,diego,bpalop}@infor.uva.es
[2] Dpto. Física y Matemáticas, Universidad de Alcalá,
Alcalá de Henares, Madrid, Spain
david.orden@uah.es

Abstract. Scheduling is one of the factors that most directly affect performance in Thread-Level Speculation (TLS). Since loops may present dependences that cannot be predicted before runtime, finding a good chunk size is not a simple task. The most used mechanism, Fixed-Size Chunking (FSC), requires many "dry-runs" to set the optimal chunk size. If the loop does not present dependence violations at runtime, scheduling only needs to deal with load balancing issues. For loops where the general pattern of dependences is known, as is the case with Randomized Incremental Algorithms, specialized mechanisms have been designed to maximize performance. To make TLS available to a wider community, a general scheduling algorithm that does not require a-priori knowledge of the expected pattern of dependences nor previous dry-runs to adjust any parameter is needed. In this paper, we present an algorithm that estimates at runtime the best size of the next chunk to be scheduled. This algorithm takes advantage of our previous knowledge in the design and test of other scheduling mechanisms, and it has a solid mathematical basis. The result is a method that, using information of the execution of the previous chunks, decides the size of the next chunk to be scheduled. Our experimental results show that the use of the proposed scheduling function compares or even increases the performance that can be obtained by FSC, greatly reducing the need of a costly and careful search for the best fixed chunk size.

Keywords: Thread-level speculation · Speculative parallelization · Speculative multithreading · Scheduling

1 Introduction

Thread-Level Speculation (TLS) [4,18,20] is the most promising technique for automatic extraction of parallelism of irregular loops. With TLS, loops that can not be analyzed at compile time are optimistically executed in parallel. A hardware or software mechanism ensures that all threads access to shared data according to sequential semantics. A *dependence violation* appears when

J.L. Träff et al. (Eds.): Euro-Par 2015, LNCS 9233, pp. 135–146, 2015.
DOI: 10.1007/978-3-662-48096-0_11

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. [10, 20]) interrupt the speculative execution and re-execute the loop serially. Subsequent approaches [5, 7, 21] squash only the offending thread and its successors, re-starting them with the correct data values. More sophisticate solutions [9, 15, 22] squash only the offending thread and subsequent threads that have actually consumed any value from it.

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

A widely used mechanism to solve this problem is to choose a fixed, optimum size by trial and error. This method, called Fixed-Size Chunking [12] requires many dry-runs to find an acceptable value. Moreover, a particular size found for one application is of little use for another one, or even for a different input set of the same application.

In this work we address the general problem of scheduling chunks of iterations for their speculative execution, regardless of the number of dependence violations that may actually appear. We have found that the pattern of dependence violations heavily depends on the application. Therefore, a scheduling strategy that is able to dynamically adapt the size of chunks at runtime is very desirable.

In this paper, we introduce a scheduling method, called *Moody Scheduling*, that tries to predict, at runtime, the best chunk size for the next chunk to be scheduled. To do so, we rely on the number of re-executions of the previous chunks, not only by using the mean of the last re-executions, but also their tendency. With this method, we are able to (a) provide a general solution that does not need an in-depth study of the dependence violation pattern, and (b) greatly reduce the need of repetitive executions to tune the scheduling mechanism used.

The rest of the paper is organized as follows. Section 2 reviews some of the existent scheduling alternatives currently used with TLS. Section 3 introduces the main aspects of our proposal. Section 4 describes the function from a mathematical point of view. Section 5 explores two different uses for our Moody Scheduling. Section 6 gives some experimental results, comparing the new algorithm with FSC, while Sect. 7 concludes this paper.

2 Related Work

Since the size of the chunk assigned to each processor directly affects performance in TLS, numerous algorithms have been proposed to give a solution to this problem. The simplest one, called Fixed-Size Chunking (FSC), was initially proposed by Kruskal and Weiss [12]. With this mechanism, each thread is assigned a constant number of iterations. Finding the right constant needs several dry-runs on each particular input set for each parallelized loop. When no dependence violations arise at runtime, this technique is perfectly adequate. The only remaining concern is to achieve a good load balance when the last iterations are being scheduled. Some examples of mechanisms that implement load-balancing techniques can be found in [11] or [23].

There are solutions based on compile-time dependence analysis [19,25]. In these approaches, scheduling decisions are taken by reviewing the possible dependence pattern that can arise, so an in-depth analysis of the loop is needed.

Other approaches rely on the *expected* dependence pattern of the loop to be parallelized. In particular, for Randomized Incremental Algorithms, where dependences tend to accumulate in the first iterations of the loop, two methods have been shown to improve performance. The first one, called Meseta [16], divides the execution in three stages. In the first one, chunks of increasing sizes are scheduled, aiming to compensate for possible dependence violations, until a lower bound of the probability of finding a dependence is reached. From then on, a second stage applies FSC to execute most of the remaining iterations. A third stage gradually decreases the chunk size, aiming to achieve a better load balancing.

The second mechanism is called Just-In-Time (JIT) Scheduling [17]. This method also focuses on randomized incremental algorithms, where dependences are more likely to appear during the execution of the first chunks. JIT Scheduling defines different logarithmic-based functions that issue chunks of increasing size, and relies on runtime information to modulate these functions according to the number of dependence violations that effectively appear.

Kulkarni et al. [13] also discussed the importance of scheduling strategies in TLS. These authors defined a schedule through three steps, i.e., three design choices that specify the behavior of a schedule, namely *clustering*, *labeling* and *ordering*. They tested several strategies for each defined module, using their Galois framework. Their results show that each application analyzed was closely linked to a different scheduling strategy.

In summary, we can conclude that proposed solutions so far either depend on the expected dependency pattern of the loop to be speculatively executed, or require a big number of training experiments to be tuned, as in the case of FSC. In this paper we present a new mechanism that issues chunks of different sizes, by taking into account the actual occurrence of dependence violations, without using any prior knowledge about their distribution.

3 Moody Scheduling: Design Guidelines

Our main purpose is to design a scheduling function that is able to predict the best size for the following chunk to be issued at runtime, without the need of a knowledge of the underlying problem. In order to decide the size of the

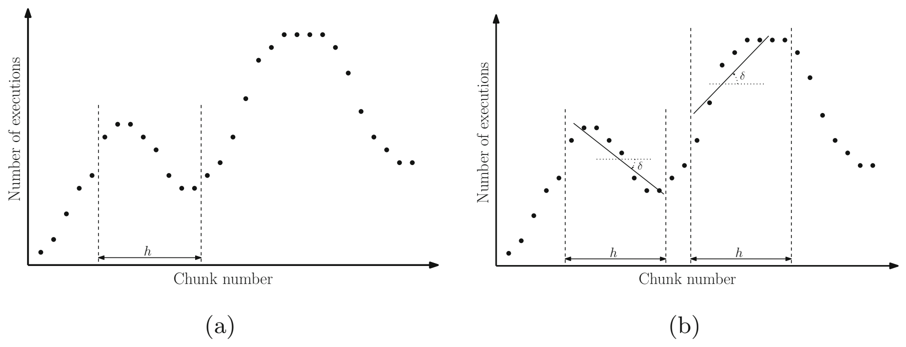

Fig. 1. (a) A possible execution profile for a given loop, and (b) an example of the use of linear regression to measure the tendency of the last h chunks. Recall that the y-axis does not represent the chunk size, but the number of re-executions for each chunk.

next chunk to be scheduled, we will use the number of times that the last h chunks have been squashed and re-executed due to dependence violations. As an example, Fig. 1(a) shows, for each scheduled chunk (x-axis), the number of times it has been executed so far (y-axis).

Given the number of executions of the last h chunks (regardless whether they were already committed or not), we will consider two parameters. The first one is the average number of executions of the last h chunks, which we call meanH and whose value is, at least, 1. The second one is the *tendency* of these re-executions. This value, which we call d, lies in the interval $(-1, 1)$ and determines if the number of executions is decreasing ($d < 0$), increasing ($d > 0$), or remaining unchanged ($d = 0$). As we will see, d depends on the angle δ between the linear regression line for the last h chunks and the horizontal axis (see Fig. 1(b)).

The size of the following chunk to be scheduled will depend on these two parameters. We will first present an informal description of the idea. The following section shows the mathematical background and the implementation details.

1. If the tendency of re-executions is decreasing (d close to -1):
   (a) If meanH is very low (close to 1), we will (optimistically) set the chunk size to the maximum size suitable for this problem. We will call this maximum value maxChunkSize.
   (b) If meanH is between the minimum value (1) and an *acceptable* value (that we call accMeanH), we will (optimistically) increase the chunk size.
   (c) If meanH is between accMeanH and an upper limit (that we call maxMeanH), we will keep the same chunk size, with the aim that its execution will help to further reduce meanH.
   (d) If meanH is higher than maxMeanH, we set the size of the following chunk to 1.
2. If the tendency of re-executions is stable (d close to 0):
   (a) If meanH is very low (close to 1), then we will (optimistically) issue a larger chunk size.

   (b) If meanH is acceptable (close to accMeanH), then we will keep the same chunk size.
   (c) If meanH is between accMeanH and maxMeanH, then we will (pessimistically) decrease the chunk size.
   (d) If meanH is higher than maxMeanH, we set the size of the following chunk to 1.
3. If the tendency of re-executions is increasing (d close to 1):
   (a) If meanH is very low (close to 1), then we propose to keep the same chunk size, waiting for the next data to confirm if meanH really gets larger.
   (b) If meanH is acceptable (close to accMeanH), then we decrease the chunk size, intending to reduce the number of executions.
   (c) If meanH is close to (or higher than) maxMeanH, then we propose a chunk of size 1 intending to minimize the number of re-executions.

The last question is what size we should use to issue the first chunk, where there is no past history to rely on. As we will see in Sect. 6, setting this inital value to 1 leads to a good performance in all the applications considered.

Table 1 summarizes the behavior of our scheduling mechanism. Using this approach, given the current lastChunkSize and a pair of values (d, meanH) our function will use the guidelines described above to propose a value for nextChunkSize. The following section discusses the implementation details.

Table 1. Changes on the following chunk sized according to d and meanH parameters

	meanH $\approx$ 1	meanH $\approx$ accMeanH	meanH $\approx$ maxMeanH	meanH $>$ maxMeanH
$d \rightarrow -1$	$\uparrow$	$\nearrow$	$=$	1
$d \approx 0$	$\nearrow$	$=$	$\searrow$	1
$d \rightarrow 1$	$=$	$\searrow$	1	1

4 Moody Scheduling Function Definition

After the informal description presented above, the following step is to define a function that determines the value for nextChunkSize using the current value of lastChunkSize, together with d and meanH. In order to obtain the value of δ, we compute the regression line defined by the last h points in our execution window (see Fig. 1(b)).

The main problem with the intuitive behavior described above is that its straightforward implementation (with nested if...then constructs) leads to a discontinuous function. This is not a desirable situation, since the behavior of the scheduling function would drastically change for very similar situations.

Instead, we define a bidimensional function that, for a given value of meanH and d, returns the size of the next chunk to be scheduled. Figure 2(a) shows a 3D

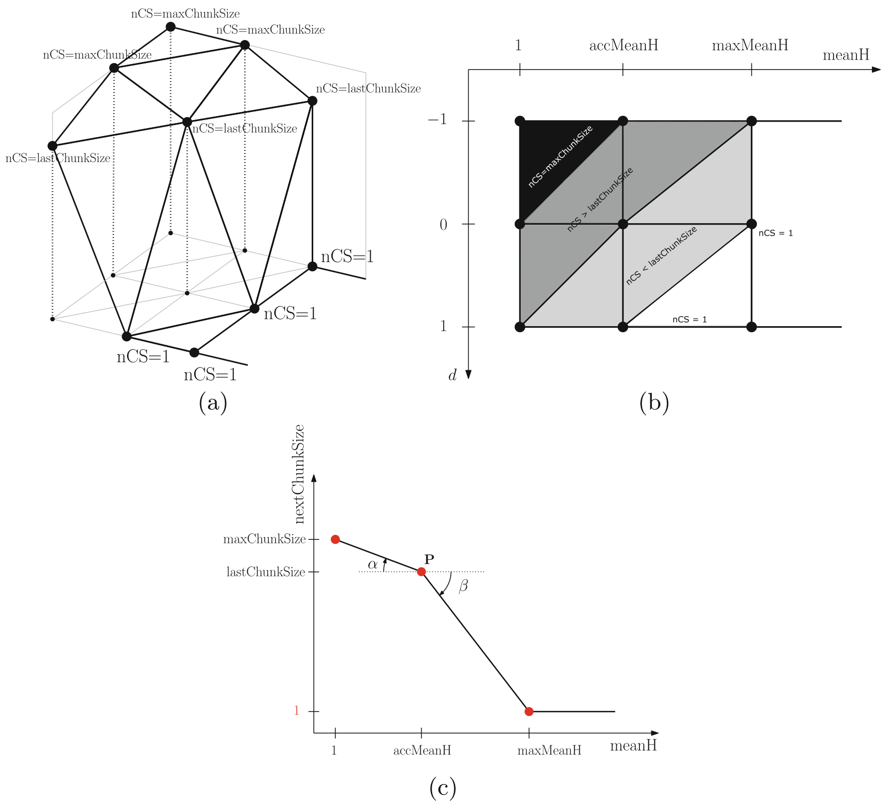

Fig. 2. (a) 3D representation of the Moody Scheduling function, that returns a value for nextChunkSize (nCS) provided the current lastChunkSize and depending on d and meanH; (b) 2D representation that connects our function with the intuitive behavior described in Sect. 3; (c) Intersection of the graphic of nextChunkSize (d, meanH) with $d = 0$.

representation of the Moody Scheduling function proposed. Figure 2(b) shows its projection onto a horizontal plane, using the same grey scale as in Table 1.

To properly define this scheduling function, several parameters should be set. The value of d is calculated by measuring the angle δ of the tendency with respect to the horizontal axis. This angle lies in $(-\pi/2, \pi/2)$. Our growth tendency $d \in (-1, 1)$ will be given by $d = \frac{\delta}{\pi/2}$.

The following parameter to be defined is accMeanH, that is, the highest value of meanH considered to be acceptable. We initially set accMeanH $= 2$, considering that, on average, we will accept that chunks have to be reexecuted at most once.

There are two remaining parameters: maxChunkSize and maxMeanH, whose values depend on the slopes of the graphic of the bidimensional scheduling function as follows. If we fix $d = 0$ in the scheduling function, we obtain the plot depicted in Fig. 2(c). In this case, we can define two angles, α and β (see figure). The angle α represents how optimistically the chunk size is going to be increased. The higher the value for α, the most optimistic the scheduling function will be. Analogously, β represents how pessimistically the chunk size is going to be decreased. If we fix the value for these two angles, the value of maxChunkSize is determined by the intersection between the segment from P with angle α, and the vertical line defined by meanH = 1. Analogously, the value for maxMeanH is determined by the intersection between the segment from P with angle β, and the horizontal line defined by nextChunkSize = 1. In the case that lastChunkSize = 1, β will be 0. On the other hand, $\alpha \neq 0$ as long as accMeanH will never be set to 1.

The nine particular points defined by meanH $\in \{1, \text{accMeanH}, \text{maxMeanH}\}$ and $d \in \{-1, 0, 1\}$ are defined by the values described above. Given that the call to nextChunkSize(d, meanH) will return maxChunkSize for the three points $(-1, 1)$, $(-1, \text{accMeanH})$, and $(0, 1)$, the function will also return maxChunkSize to all points inside this triangle. Analogously, for all points inside the triangle with vertices $(1, \text{accMeanH})$, $(1, \text{maxMeanH})$, and $(0, \text{maxMeanH})$, the function will return 1. Notice that points on the diagonals $(1, 1)$ to $(0, \text{accMeanH})$, and from there to $(-1, \text{maxMeanH})$ will return lastChunkSize. These three facts provide a natural triangulation for the space in Fig. 2(b).

5 Dynamic and Adaptive Implementations

If no dependences arose during the parallel execution, the size of the following chunk would be calculated only once, that is, just before issuing its execution. Otherwise, if the execution of the chunk fails, it gives the runtime system an opportunity to adjust its calculation by calling the scheduling function with updated runtime information. As it happens in [17], this leads to two different ways to use the scheduling function:

– To calculate the size of the following chunk only the first time this particular chunk will be issued. Subsequent re-executions will keep the same size. See Fig. 3(a).
– To re-calculate the size of the following chunk each time the chunk is scheduled. This solution is called *adaptive scheduling* in [17]. See Fig. 3(b).

The advantage of adaptive over dynamic scheduling is that the first calculation of the chunk size may rely on incomplete information, since some or all of the previous chunks are still being executed, and therefore they may suffer additional squashes. Adaptive scheduling will always reconsider the situation using updated data. Naturally, this comes at the cost of additional calls to the scheduling function.

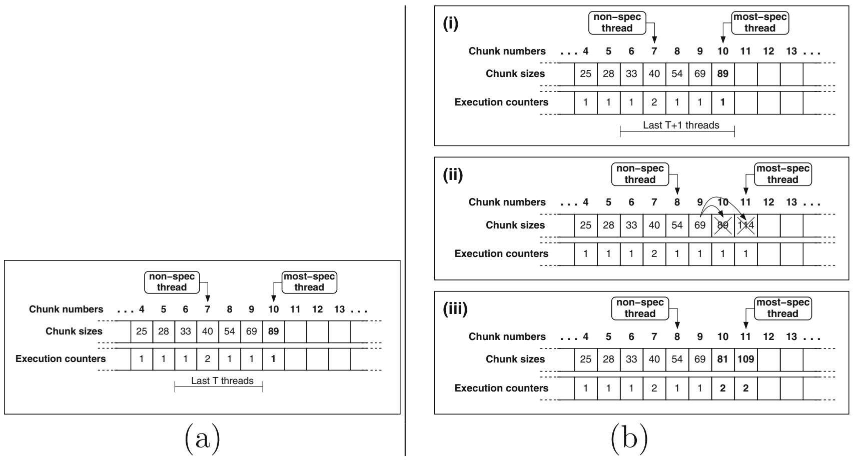
non–spec thread
most–spec thread
Chunk numbers . . . 4 5 6 7 8 9 10 11 12 13 . . .
Chunk sizes 25 28 33 40 54 69 89
Execution counters 1 1 1 2 1 1 1
Last T threads
(a)
(i)
Chunk sizes 25 28 33 40 54 69 89
Execution counters 1 1 1 2 1 1 1
Last T+1 threads
(ii)
Chunk sizes 25 28 33 40 54 69 89 114
Execution counters 1 1 1 2 1 1 1 1
(iii)
Chunk sizes 25 28 33 40 54 69 81 109
Execution counters 1 1 1 2 1 1 2 2
(b)

Fig. 3. (a) Dynamic Moody Scheduling. The size for the following chunk to be executed (#10) is calculated once (89 iterations). Its size will be preserved regardless of the number of re-executions of this chunk. (b) Adaptive Moody Scheduling. (i) Size of chunk #10 is calculated with the Moody Scheduling function (89 iterations). (ii) Chunk #9 issues a squash operation. (iii) Squashed threads recalculate in program order the new sizes of the chunks to be executed, using the new values of the execution counters.

6 Experimental Evaluation

We have used ATLaS, a software-based TLS framework [1,8], to execute in parallel four different applications that present non-analyzable loops with and without dependences among iterations.

The first benchmark used is TREE from [2]. This application spends a large fraction of its sequential execution time on a loop that can not be automatically parallelized by state-of-the-art compilers because it has dependence structures that are either too complicated to be analyzed at compile time or dependent on the input data.

We consider three additional applications that present loops with dependences. The first one is the 2-Dimensional Convex Hull (2D-Hull), an incremental randomized algorithm due to Clarkson et al. [6]. The algorithm computes the convex hull (smallest enclosing convex polygon) of a set of two-dimensional points in the plane. We have tested this application using three different input sets: Disc and Square, that are composed of points uniformly distributed inside a disc and a square, and Kuzmin, that is composed of points that follow a Kuzmin distribution [3].

The second application, called the 2-Dimensional Minimum Enclosing Circle (2D-MEC) [24], finds the smallest enclosing circle containing a given set of points in the plane. The construction is also incremental. In this case, a dependence violation forces not only an update of the current solution, but the recalculation of the entire enclosing ball. This fact produces devastating effects when the benchmark is speculatively parallelized.

Table 2. Characteristics of the algorithms and input sizes used

Algorithm	Input set description	Loop parallelized	Loop time as % of total time	Iterations per invocation	% of dependence violations	FSC chunk size used (iterations)
`TREE`	Off-axis parab. collision	accel_10	94	4 096	0	100
`2D-Hull`	Kuzmin, 10M points	Main loop	99	9 999 997	0.0008	11 000
`2D-Hull`	Square, 10M points	Main loop	99	9 999 997	0.0032	3 000
`2D-Hull`	Disc, 10M points	Main loop	99	9 999 997	0.021	1 250
`2D-MEC`	Disc, 10M points	Inner loop	99	Changes dynamically	0.009	1 800
`Delaunay`	100K points	Main loop	99	95 000	0.5	2

The last benchmark is the Delaunay triangulation [14] of a two-dimensional set of points. We have used an input set of 100 K points. Table 2 summarizes the characteristics of each application considered.

Experiments were carried out on a 64-processor server, equipped with four 16-core AMD Opteron 6376 processors at 2.3 GHz and 256 GB of RAM, which runs Ubuntu 12.04.3 LTS. All threads had exclusive access to the processors during the execution of the experiments, and we used wall-clock times in our measurements. Applications were compiled with gcc. Times shown below 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.

Figure 4 shows the relative performance of the mentioned applications when executed with the ATLaS speculative parallelization framework [1] and three different scheduling mechanisms: Adaptive Moody Scheduling, Dynamic Moody Scheduling and Fixed-Size Chunking (FSC).

The plots show the performance obtained when an optimum chunk size is used for FSC (a choice that required more than 20 experiments per application) and for Moody Scheduling, whose choice of parameters required less than five experiments in all cases. In the case of Moody Scheduling, we have used a value of 2 for accMeanH, $\beta = \frac{\pi}{4}$, and a value for h (the size of the window to be considered) equal to twice the number of processors for all applications. Regarding α, we have used values $\in (\frac{\pi}{20}, \frac{\pi}{6})$, depending on whether the application is known to produce dependence violations at runtime.

Furthermore, Moody Scheduling turns out to be competitive even without any tuning: If we set to 1 the initial chunk size, its performance reaches 88.3 % of the best FSC on geometric average. Meanwhile, the performance of FSC with chunk size 1 drops almost to zero (except for Delaunay, when the best chunk size for FSC is 2).

Regarding 2D-Hull (Fig. 4(a), (b), and (c)), the results for the Disc and Square input sets show that our scheduling method leads to a better performance than FSC. For the Disc input set, the highest speedup (2.17×) is achieved with 32 processors and the Dynamic version. For the Square input set, the biggest speedup (6.81×) is achieved with the Dynamic version and 40 processors. Finally, the performance figures when processing the Kuzmin input set are similar for all the scheduling alternatives. The best performance (11.11×) is achieved with 56 processors and the Adaptive version. The two remaining applications lead

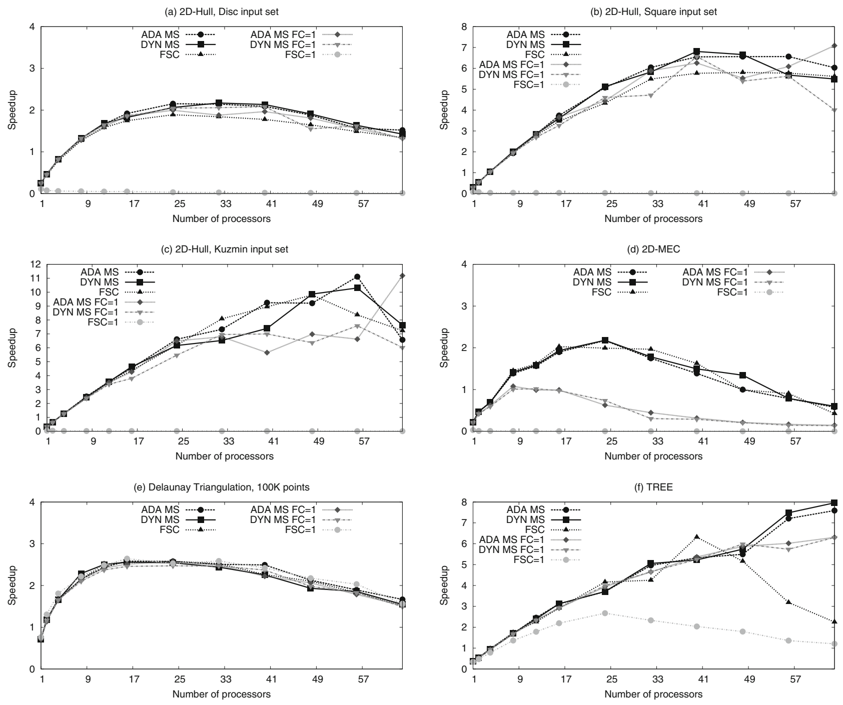

Fig. 4. Performance comparison for 2D-Hull with Disc, Square, and Kuzmin input sets, and 2D-MEC, Delaunay, and TREE benchmarks. Note the extremely poor performance of FSC when the chunk size is set to 1.

to similar performance results with all the scheduling mechanisms considered. The 2D Smallest Enclosing Circle (Fig. 4(d)) achieved a speedup of 2.18× with 24 processors and the Dynamic version. The Delaunay triangulation (Fig. 4(e)) achieved a speedup of 2.58× using the Adaptive version. Finally, Fig. 4(f) shows the speedup of TREE. This benchmark gained with the use of our scheduling method: With 40 processors, FSC approach achieved its speedup peak, while the Adaptive version continued improving its performance even with the maximum of available processors. It is interesting to note that, for TREE, both the Dynamic and Adaptive mechanisms are equivalent: As long as no squashes are issued, the size for each new chunk is calculated only once. The best performance in this benchmark (7.96×) is obtained with the Adaptive version and 64 processors.

Regarding the relative performance of FSC and Moody Scheduling, both strategies lead to similar performance figures for all applications, with the exception of the TREE benchmark, where Moody Scheduling is clearly better. The main difference between them is that the choice of the optimum block size in FSC required a prior, extensive testing (more than 20 runs per benchmark), while the Moody Scheduling self-tuning mechanism leads to competitive results right from the beginning. Moreover, the results obtained for the TREE application show that, contrary to intuition, our self-tuning mechanism leads to better results than FSC even without dependence violations, despite the higher computing cost added by the runtime calls to the Moody Scheduling function. Regarding which approach is better, Dynamic or Adaptive, it seems to depend on the application. Therefore, we will keep both of them in the ATLaS framework.

7 Conclusions

This work addresses an important problem for speculative parallelism: How to compute the size of the following chunk of iterations to be scheduled. We have found that most of the existent solutions are highly dependent on the particular application to parallelize, and they require many executions of the problem to obtain the scheduling parameters. Our new method, Moody Scheduling, automatically calculates an adequate size for the next chunk of iterations to be scheduled, and can be tuned further by making slight changes to its parameters, namely α, β, h, and accMeanH. Our scheduling method can be used as a general approach that avoids most of the 'dry-runs' required to arrive to scheduling parameters in other methods. Results show that execution times are similar (or better) to those obtained with a carefully-tuned FSC execution. Moody Scheduling just needs from the user to decide how optimistic, and pessimistic, the TLS system will be when it schedules the following chunk of iterations.

Acknowledgments. The authors would like to thank the anonymous referees for their comments. This research is partly supported by Castilla-Leon (VA172A12-2), MICINN (Spain) and the European Union FEDER (MOGECOPP project TIN2011-25639, HomProg-HetSys project, TIN2014-58876-P, CAPAP-H5 network TIN2014-53522-REDT), Madrid Regional Government through the TIGRE5-CM program (S2013/ICE-2919), and by the MICINN Project MTM2011-22792. Belen Palop is partially supported by MINECO MTM2012-30951.

References

1. Aldea, S., Estebanez, A., Llanos, D.R., Gonzalez-Escribano, A.: An OpenMP extension that supports thread-level speculation. IEEE Trans. Parallel Distrib. Syst. (2015, to appear)
2. Barnes, J.E.: Institute for Astronomy, University of Hawaii. ftp://ftp.ifa.hawaii.edu/pub/barnes/treecode/
3. Blelloch, G.E., Miller, G.L., Hardwick, J.C., Talmor, D.: Design and implementation of a practical parallel delaunay algorithm. Algorithmica **24**(3), 243–269 (1999)
4. Cintra, M., Llanos, D.R.: Toward efficient and robust software speculative parallelization on multiprocessors. In: Proceedings of the PPoPP 2003, pp. 13–24. ACM (2003)
5. Cintra, M., Llanos, D.R.: Design space exploration of a software speculative parallelization scheme. IEEE Trans. Parallel Distrib. Syst. **16**(6), 562–576 (2005)
6. Clarkson, K.L., Mehlhorn, K., Seidel, R.: Four results on randomized incremental constructions. Comput. Geom. Theor. Appl. **3**(4), 185–212 (1993)

7. Dang, F.H., Yu, H., Rauchwerger, L.: The R-LRPD test: speculative parallelization of partially parallel loops. In: Proceedings of the 16th IPDPS, pp. 20–29. IEEE Computer Society (2002)
8. Estebanez, A., Llanos, D., Gonzalez-Escribano, A.: New data structures to handle speculative parallelization at runtime. International Journal of Parallel Programming pp. 1–20 (2015)
9. García-Yágüez, A., Llanos, D.R., Gonzalez-Escribano, A.: Squashing alternatives for software-based speculative parallelization. IEEE Trans. Comput. **63**(7), 1826–1839 (2014)
10. Gupta, M., Nim, R.: Techniques for speculative run-time parallelization of loops. In: Proceedings of the ICS 1998, pp. 1–12. IEEE Computer Society (1998)
11. Hagerup, T.: Allocating independent tasks to parallel processors: an experimental study. J. Parallel Distrib. Comput. **47**(2), 185–197 (1997)
12. Kruskal, C., Weiss, A.: Allocating independent subtasks on parallel processors. IEEE Trans. SE- Softw. Eng. **11**(10), 1001–1016 (1985)
13. Kulkarni, M., Carribault, P., Pingali, K., Ramanarayanan, G., Walter, B., Bala, K., Chew, L.P.: Scheduling strategies for optimistic parallel execution of irregular programs. In: Proceedings of the 20th SPAA, pp. 217–228. ACM (2008)
14. Lee, D., Schachter, B.: Two algorithms for constructing a delaunay triangulation. Int. J. Comput. Inf. Sci. **9**(3), 219–242 (1980)
15. Li, X.F., Du, Z., Yang, C., Lim, C.C., Ngai, T.F.: Speculative parallel threading architecture and compilation. In: Proceedings of the ICPPW 2005, pp. 285–294. IEEE Computer Society (2005)
16. Llanos, D.R., Orden, D., Palop, B.: Meseta: a new scheduling strategy for speculative parallelization of randomized incremental algorithms. In: HPSEC-05 Workshop (ICPP 2005), pp. 121–128. IEEE Computer Society, Oslo, June 2005
17. Llanos, D.R., Orden, D., Palop, B.: Just-in-time scheduling for loop-based speculative parallelization. In: PDP 2008, pp. 334–342 (2008)
18. Oancea, C.E., Mycroft, A., Harris, T.: A lightweight in-place implementation for software thread-level speculation. In: Proceedings of the SPAA 2009, pp. 223–232. ACM (2009)
19. Ottoni, G., August, D.: Global multi-threaded instruction scheduling. In: Proceedings of the MICRO 40, pp. 56–68. IEEE Computer Society, Washington, DC, USA (2007)
20. Rauchwerger, L., Padua, D.: The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In: Proceedings of the PLDI 1995, pp. 218–232. ACM (1995)
21. Rundberg, P., Stenström, P.: An all-software thread-level data dependence speculation system for multiprocessors. J. Instr.-Level Parallelism **2001**(3), 1–26 (2001)
22. Tian, C., Feng, M., Gupta, R.: Speculative parallelization using state separation and multiple value prediction. In: Proceedings of the 2010 International Symposium on Memory Management, ISMM 2010, pp. 63–72. ACM, New York (2010)
23. Tzen, T.H., Ni, L.M.: Trapezoid self-scheduling: a pratical scheduling scheme for parallel compilers. IEEE Trans. Parallel Distrib. Syst. **4**(1), 87–98 (1993)
24. Welzl, E.: Smallest enclosing disks (balls and ellipsoids). In: Maurer, H.A. (ed.) New Results and New Trends in Computer Science. LNCS, vol. 555, pp. 359–370. Springer, Heidelberg (1991)
25. Zhai, A., Steffan, J.G., Colohan, C.B., Mowry, T.C.: Compiler and hardware support for reducing the synchronization of speculative threads. ACM Trans. Archit. Code Optim. **5**(1), 3–33 (2008)