Int J Parallel Prog (2017) 45:225–241
DOI 10.1007/s10766-016-0421-x

Using the Xeon Phi Platform to Run Speculatively-Parallelized Codes

Alvaro Estebanez[1] · **Diego R. Llanos**[1] · **Arturo Gonzalez-Escribano**[1]

Received: 14 July 2015 / Accepted: 30 March 2016 / Published online: 8 April 2016

Abstract Intel Xeon Phi accelerators are one of the newest devices used in the field of parallel computing. However, there are comparatively few studies concerning their performance when using most of the existing parallelization techniques. One of them is thread-level speculation, a technique that optimistically tries to extract parallelism of loops without the need of a compile-time analysis that guarantees that the loop can be executed in parallel. In this article we evaluate the performance delivered by an Intel Xeon Phi coprocessor when using a software, state-of-the-art thread-level speculative parallelization library in the execution of well-known benchmarks. We describe both the internal characteristics of the Xeon Phi platform and the particularities of the thread-level speculation library being used as benchmark. Our results show that, although the Xeon Phi delivers a relatively good speedup in comparison with a shared-memory architecture in terms of scalability, the relatively low computing power of its computational units when specific vectorization and SIMD instructions are not fully exploited makes this first generation of Xeon Phi architectures not competitive (in terms of absolute performance) with respect to conventional multicore systems for the execution of speculatively parallelized code.

Keywords Thread-level speculation · Speculative parallelization · Optimistic parallelization · Xeon Phi

✉ Diego R. Llanos
diego@infor.uva.es

Alvaro Estebanez
alvaro@infor.uva.es

Arturo Gonzalez-Escribano
arturo@infor.uva.es

[1] Departamento de Informatica, Universidad de Valladolid, Campus M. Delibes, 47011 Valladolid, Spain

1 Introduction

Currently, physical limitations of single core chips are inducing a quick development of multicore architectures. One of the most recent approaches is the Intel®Xeon Phi™ [3,11,20], a coprocessor with more than 60 cores able to execute both offloaded and native codes. Nonetheless, due to its own novelty, this coprocessor has not yet been extensively tested with non-regular parallel codes. The dissemination of experimental results under these conditions would be really useful to test the behavior and capabilities of this computing resource.

In this paper we use a Xeon Phi coprocessor to run irregular applications that were speculatively parallelized, with the help of a software-only, speculative parallelization library. Thread-level speculation (TLS) [8,34,35,42], also called speculative parallelization (SP) [14,18,22,46] or optimistic parallelism [25,26] tries to extract parallelism of loops that can not be considered fully parallel at compile time. TLS optimistically assumes that dependence violations will not occur, launching the parallel execution of the loop. A hardware or software monitor ensures the correctness of that assumption. If a dependence violation is detected, offending threads are stopped and re-started in order. After solving the issue, the optimistic, parallel execution is allowed to continue. The target of TLS systems are usually for loops. Other loops can be considered as well, but as long as their number of iterations can not be so easily predicted, the applicability of TLS solutions is limited by scheduling problems.

In order to handle the speculative parallelization of a loop, all variables have to be classified as private, shared, or "speculative".[1] All reads to a speculative variable are replaced at compile time with a function that recovers the most up-to-date value for this variable. In a similar way, all writes to a speculative variable are replaced with a function that not only performs the write operation, but also ensures that no thread executing a subsequent iteration has already consumed an outdated value of this variable. TLS is useful when executing codes that present scarce dependence violations at runtime. Otherwise, costs associated to check for correctness, stop and retry executions, and commitments, make this technique inefficient.

The contribution of this paper is to test the performance of a state-of-the-art TLS runtime library using an Intel Xeon Phi. This coprocessor has a big number of parallel threads, therefore, it is interesting to measure its behavior with a shared-memory technique such as TLS, when data is permanently shared among threads. We believe that TLS is also a good problem to test such hardware architecture, because the Xeon Phi platform allows the different threads to follow different execution paths (contrary to GPUs), so the use of a Xeon Phi platform can be viewed as a natural environment for TLS. Our experimental results show that the benchmarks considered scale well when running them on a Xeon Phi coprocessor. However, our results also confirm that, due to the irregular nature of the target applications for TLS techniques, and the modest computing capabilities of each individual core when vectorized and SIMD instructions are not fully exploited, execution times are much higher than those gauged in conventional shared-memory systems. This limit the usability of the current genera-

[1] This issue can be addressed by the programmer, or by the use of specific compilers such as [4].

tion of Xeon Phi platform for irregular, not-easily-vectorized applications, although we expect that this situation will change with the new generation of the Xeon Phi platform, that will incorporate out-of-order, more powerful processors.

The rest of this paper is structured as follows: Sect. 2 describes the main characteristics of the Xeon Phi coprocessor. Section 3 describes the software-based, TLS framework used to test the coprocessor. Section 4 describes both the experimental environment and the benchmarks used. Section 5 shows some experimental results in terms of performance measured in a shared-memory system without coprocessor, and in a Xeon Phi coprocessor. Section 6 summarizes some works that helps to put into perspective our contribution. Finally, Sect. 7 concludes this paper.

2 Intel Xeon Phi in a Nutshell

Intel Xeon Phi [3,11,20] is a coprocessor launched by Intel in 2012. It is called coprocessor because, although it can run a Linux operating system by itself, it should be placed aside another processor to work properly. Although first impressions might suggest a number of similarities, it is not an accelerator such as GPUs. Whereas the Intel Xeon Phi cores are more similar to classical complete CPUs, the GPUs thread scheduling hardware is different. As the reader may know, GPUs have a hierarchical hardware architecture, so they should be programmed with a hierarchical thread structure in mind [7], that uses the concept of threads, blocks, and grids.[2] Furthermore, Intel Xeon Phi coprocessors do not use the grid, and groups of threads in the same way, and also the memory latency hiding mechanisms are different. This issue hinders easy code migrations to the latter kind of accelerators, and requires and in-depth understanding of special programming models as CUDA [30], or OpenCL [23]. On the other hand, the Xeon Phi coprocessor is able to use all standard parallel programming models such as OpenMP [12], POSIX threads, MPI [43], or even OpenCL. Thus, using this new coprocessor only requires a minimum learning curve, assuming that the programmer knows at least one of these common parallel programming models.

2.1 Internal Details

Intel Xeon Phi coprocessors have up to 61 cores at 1090 MHz, interconnected by a high-speed bidirectional ring. Each core is enhanced with four hardware threads (up to 244 threads per coprocessor), and with a 512-KB L2 cache. L2 cache levels are shared by all cores. Furthermore, in addition to 64-bit x86 instructions, cores offer 512-bit wide SIMD vectors, intended to speed-up the execution of regular code through vectorization. The coprocessor is generally connected to the host system via the PCI Express bus, and supports up to 8 GB GDDR5 memory. Figure 1 briefly describes the architecture of the Intel Xeon Phi.

[2] A thread is the simplest unit of execution, intended to process a specific code. A block is defined as a group of threads, where threads can be executed concurrently or sequentially with no order. At this level, a block allow the coordination of its threads with the use of barriers. A grid is a group of blocks without any possible synchronization among them.

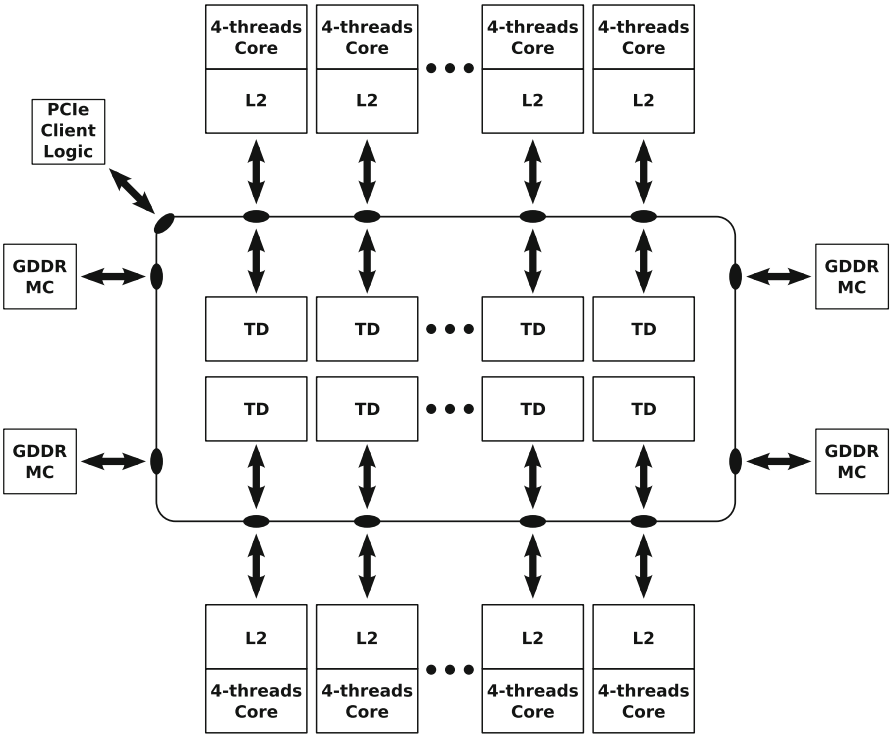

Fig. 1 Overview of the microarchitecture of an Intel Xeon Phi coprocessor

2.2 Use of the Xeon Phi

There are mainly two ways of executing a parallel program into a Xeon Phi coprocessor:

Native Execution: The Intel Xeon Phi coprocessor is capable of running a Linux operating system. It is possible to log into the Xeon Phi from the host processor using SSH, through a mic0 network interface, added to the kernel by a module provided by Intel, and use it natively. Thus, it allows the execution of the typical Linux-based commands as well as our own programs.
Offload Extensions from the host: Intel defined a set of pragmas and keywords to be used in parallel codes in order to execute them in coprocessors. A programmer only needs to declare the region which should be executed in a coprocessor. Inside this region, any kind of function can be used. For example, in the case of OpenMP, a single pragma defined as #pragma offload target mic should be used, where mic represents the identifier of the target Xeon Phi coprocessor. In addition, we should point out the variables that will be used in the coprocessor, declaring their use with the clauses in(), out(), or inout(). The use of variables with dynamic size requires to explicitly declare the size, e.g. in(a:length(n)). These variables will be copied from the host to the device, and/or vice versa, depending on their usage.

As can be seen, the Xeon Phi programming methodology is really convenient in order to gain speedup with a relatively low programming effort.

3 Description of the ATLaS Runtime Library

The ATLaS framework [4] enhances OpenMP with a new clause to allow the speculative use of variables inside a program. The use of this tool is simple: A programmer only needs to include the list of speculative variables in the predefined speculative clause of al parallel for directive. The compilation and runtime system automatically transforms the code to a version capable of running in parallel while preserving sequential semantics. To do so, the system augments all accesses to speculative variables, adapting them to the functions of the ATLaS runtime library [15], that ensures sequential consistency. In this work, we have modified the runtime library so as to adapt it to particularities of the Intel Xeon Phi coprocessor. However, our aim was doing as few modifications as possible. A more in-depth adaptation would require a deep modification of the library implementation, that is out of the scope of this paper.

The ATLaS runtime library supports the speculative execution of for loops with dynamic and pointer-referenced speculative variables, handling dynamic memory, and managing, on demand, the space needed for speculative variables in each thread. This TLS runtime library allows the parallelization of loops with variables of any data type, allowing the programmer to reference these variables either by name or by address. In this section we will briefly show the architecture of our library in order to understand the structures and operations in which the Intel Xeon Phi will be tested.

3.1 ATLaS Runtime Data Structures

Figure 2 outlines the data structures needed by the speculative runtime library. In this section we will only briefly describe the main characteristics of this solution: A detailed explanation can be found in [4,15]. At the top of the figure, we can see two pointers to the non-spec and the most-spec threads, that are the threads in charge to the execution of the non-speculative and most-speculative chunks of iterations. The non-speculative chunk is the one whose execution is not speculative, while the most-speculative one is the one that is more likely to suffer a dependence violation from a predecessor thread. Below these pointers there is a matrix with W window slots (four in the figure) implementing a sliding window that manages the runtime of the library. Each slot is responsible to handle the speculative execution of a particular set of iterations. Each slot is composed of two fields, STATE with the state of the execution being carried out in each slot; and a pointer to maintain the position of the speculative variables used by each slot in the execution.

It is very important to understand that there is not a fixed association between threads and slots. Whenever a thread is assigned a new chunk of iterations, it is also assigned a slot to work in, that is located at the right of the most-speculative

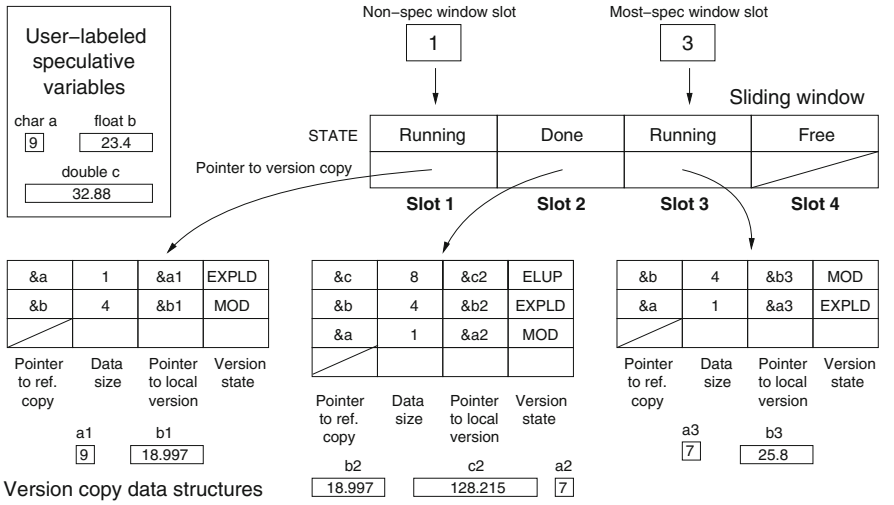

Fig. 2 Data structures of our speculative library

slot. This allows to maintain an order relationship among the chunks being executed.

In addition to its STATE, each slot points to a data structure that holds the version copies of the data being speculatively accessed. Figure 2 represents a loop with three speculative variables. At a given moment, the thread executing the non-speculative chunk has speculatively accessed variables a and b. Each row of the version copy data structure keeps the information needed to manage the access to a different speculative variable. The first column indicates the address of the original variable, known as the *reference copy*. The second one indicates the data size. The third one indicates the address of the local copy of this variable associated to this window slot. Finally, the fourth column indicates the state associated to this local copy. Once accessed by a thread, the version copies of the speculative data can be in three different states: *Exposed Loaded*, indicating that the thread has forwarded its value from a predecessor or from the main copy; *Modified*, indicating that the thread has written to that variable without having consumed its original value; and *Exposed Loaded and Updated*, where a thread has first forwarded the value for a variable and has later modified it.

Figure 2 represents a situation where the thread working in Slot 1 has performed a speculative load from variable a (obtaining its value from the reference copy) and a speculative store to variable b. Regarding a, the figure shows that the thread working in Slot 3 has forwarded its value. With respect to variable b, the information in the figure shows that b has been overwritten both by threads working in Slots 1 and 3 without taking into account its prior value (since both version are in *Modified* state). When the commitment of the data generated by these threads take place, variable b will be first overwritten by the version copy produced by the non-speculative thread. After finishing this commit operation, the non-spec pointer advances one position, and when the thread located in Slot 2 finishes, it will overwritten again the variable b with the new value.

3.2 Speculative Operations

In order to manage versioning and detect dependence violations on speculative variables, all accesses to speculative variables are replaced at compile time with a function that manages the structures described above. Reading a speculative variable implies to obtain the most updated value of this variable, in order to avoid, as much as possible, dependence violations. For the same reason, write operations are also replaced with a function that, in addition to storing the value in an intermediate place, checks if any successor thread (a thread which executes a chunk of subsequent iterations) has used an outdated version of this variable. In this case, the thread should discard the data calculated so far during the execution of the current chunk of iterations and restart it. When doing so, the thread will forward the correct value of the variable. As can be inferred, while a load or store operation of a scalar datum only requires to perform a single memory access, the transformation of this operation in a speculative load or store requires to replace the single memory access to a function call that performs all the required actions. This implies that the time consumed by the speculative load or store operation can be easily two or three orders of magnitude higher than the original one.

The partial commit operation is exclusively carried out by the non-speculative thread. Every time a thread should check if its data have to be committed or discarded, it first checks if it has not been squashed and if is the non-speculative thread. If the thread is speculative, the slot is left, since it will be committed later by the non-spec thread.

It is interesting to point out that each thread only writes on its local version copy data structure, so no critical sections are needed to protect them. The only critical section used protects the sliding window data structure, because, without it, a thread could overwrite another thread's state.

4 Experimental Setup

The goal of this work is to test the Xeon Phi coprocessor in off-loading mode to speculatively execute in parallel different, well-known benchmarks. In this way, the ATLaS runtime library was adjusted to offload the execution of the parallel loop to the Xeon Phi coprocessor, without further optimizations such as vectorization, one of the most important features of the Xeon Phi. In any case, this feature is not very useful for our benchmarks, mainly composed of irregular code.

To test the performance of the ATLaS TLS runtime, we have used three different real-world benchmarks, together with a synthetic one. The real-world applications include the 2-dimensional convex hull problem (2D-hull) [10], the Delaunay triangulation problem [13,29], and a C implementation of the TREE benchmark [5]. The synthetic benchmark is the Fast [4] benchmark.

The 2D-hull problem solves the computation of the convex hull (smallest enclosing polygon) of a set of points in the plane. We have parallelized Clarkson et al. [10]'s implementation. 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. Note that any change to the solution found so far generates a dependence violation, because other successor threads may have used the old enclosing polygon to process the points assigned to them. The probability of a dependence violation in the 2D-Hull algorithm depends on the shape of the input set. Therefore, we have used three different, ten-million-point input sets to run this benchmark. The *Kuzmin* input set follows a Gauss–Kuzmin distribution, with a higher density of points around the center of the distribution space, which leads to very few dependence violations, since points far from the center are very scarce. The two other input sets, *Square* and *Disc*, cause more dependence violations than *Kuzmin*, with their points uniformly distributed inside a square and a disc, respectively. The *Square* input set leads to an enclosing polygon with fewer edges than the *Disc* input set, thus generating fewer dependence violations.

The second real-world application is the randomized incremental construction of the Delaunay triangulation using the jump-and-walk strategy, which was introduced by Mücke et al. [13,29]. This incremental strategy starts with 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). The goal of the algorithm is to find the network of triangles in which all the circumcircles of all triangles in the network are empty, i.e., the circumcircle of each triangle contains no other vertices than those three that define the triangle. We have used an input set of 5000 anchors, and one million points to be inserted.

The TREE problem [5], unlike the previous two applications, does not suffer from dependence violations, but it is still not parallelizable at compile time because the compiler is not able to ensure that there are no data dependencies. Compilers also find hurdles in several sum and maximum reductions contained in the code. We have run this benchmark with a 4096-point input set.

We have also used a synthetic benchmark called Fast [4], which presents almost no dependences between iterations, and which was designed to test the efficiency of the speculative scheduling mechanism, with few iterations

We have used two different platforms to compare the scalability of the speculative execution of our benchmarks. The first one is Heracles, a 64-processor server, equipped with four 16-core AMD Opteron 6376 processors at 2.3 GHz and 256 GB of RAM, which runs CentOS 7 Linux. The second one is Chimera, a server equipped with two Intel Xeon E5-2620 V2 processors with six cores each, 32 Gb of RAM, and a Xeon Phi 3120A coprocessor with 6 Gb of RAM running at 1.1 GHz. The system also runs CentOS 7 Linux.

All threads had exclusive access to the processors during the execution of the experiments, and we used wall-clock times in our measurements without taking into account times spent in data transfer. We have used icc (ICC) 15.0.2 for all applications in both platforms. Although we know that ICC may be not the most appropriate compiler for an AMD platform, Xeon Phi offloaded codes can only be compiled with ICC, and we preferred to use the same compiler in all the experiments. Times shown in the following sections represent the time spent in the execution of the parallelized loop for each application. To better assess the scalability offered by the Xeon Phi, the time required for data offloading has not been taken into account in the measurements.

The execution time used as the baseline for comparing speedups was the sequential execution of each benchmark in both platforms, with the same compiler and compilation flags.

5 Experimental Results

5.1 Scalability

Figure 3 compares the speedup obtained with the same parameters in both the shared-memory processor, and the Xeon Phi coprocessor. Results show that, regarding the speedup, the Xeon Phi coprocessor delivers a better scalability than a conventional, shared-memory system. This scalability improvement is related to the Xeon Phi memory architecture. All TLS runtime libraries require many accesses to shared data, so the faster and higher bandwidth, the better performance. In our case, while the AMD Opteron 6376 achieves up to 51.2 GB/s memory bandwidth per socket, [1], the Intel Xeon Phi coprocessor achieves a peak of 240 GB/s [2]. In our experiments, the benchmark with the highest number of variables involved in the speculative execution is the

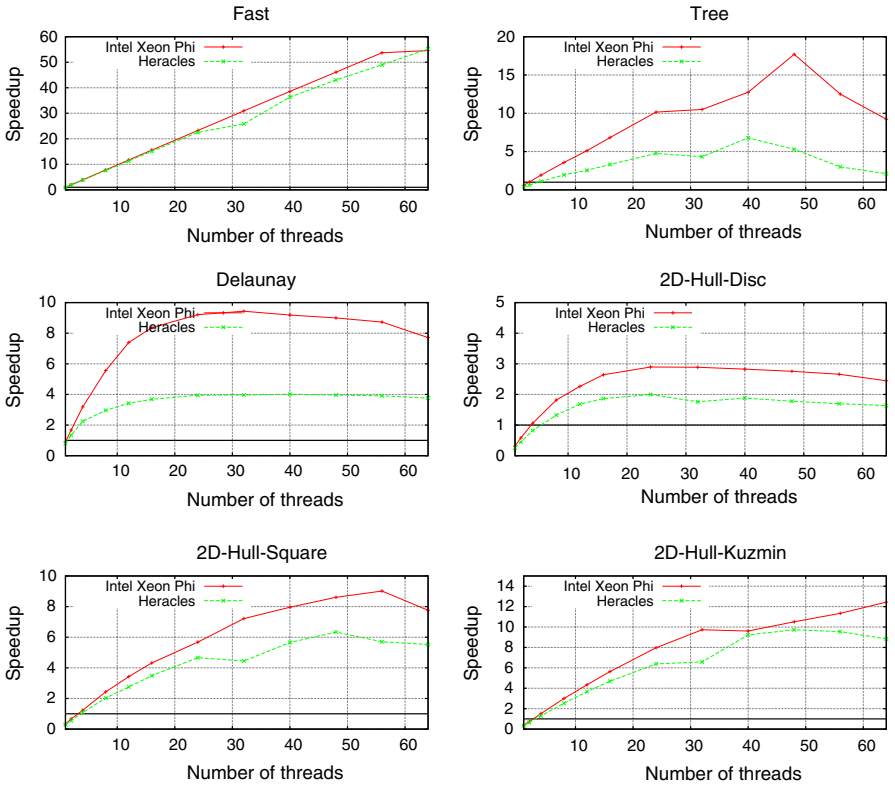

Fig. 3 Speedups by number of processors for each tested benchmark, comparing the performance obtained by using Intel Xeon Phi coprocessor, and a conventional shared-memory system

Delaunay triangulation, with more than 12 million, different scalar variables, while the one with the smallest shared data set is FAST, with just two variables. Whilst in the latter benchmark the speedup is similar in both architectures, in the Delaunay triangulation the speedup achieved by the Intel Xeon Phi is up to 2.38× higher with respect to the AMD Opteron 6376.

5.2 Oversubscription

Figure 4 shows the experimental results produced with the execution of the benchmarks using the whole threads of the Xeon Phi coprocessor. The particular nature of the threads per core in this platform, being not independent of each other, severely limits the scalability when more than 60 or 70 threads are launched, depending on the application. In some cases, performing such an oversubscription with respect to the number of cores leads to slightly better results, but the performance decays when we tried to use more cores. We attribute this fact mainly to memory issues, since the benchmarks considered mainly use integer arithmetic and therefore do not make noticeable use of FP ALU sharing among contexts. As we have exposed in Sect. 2.1,

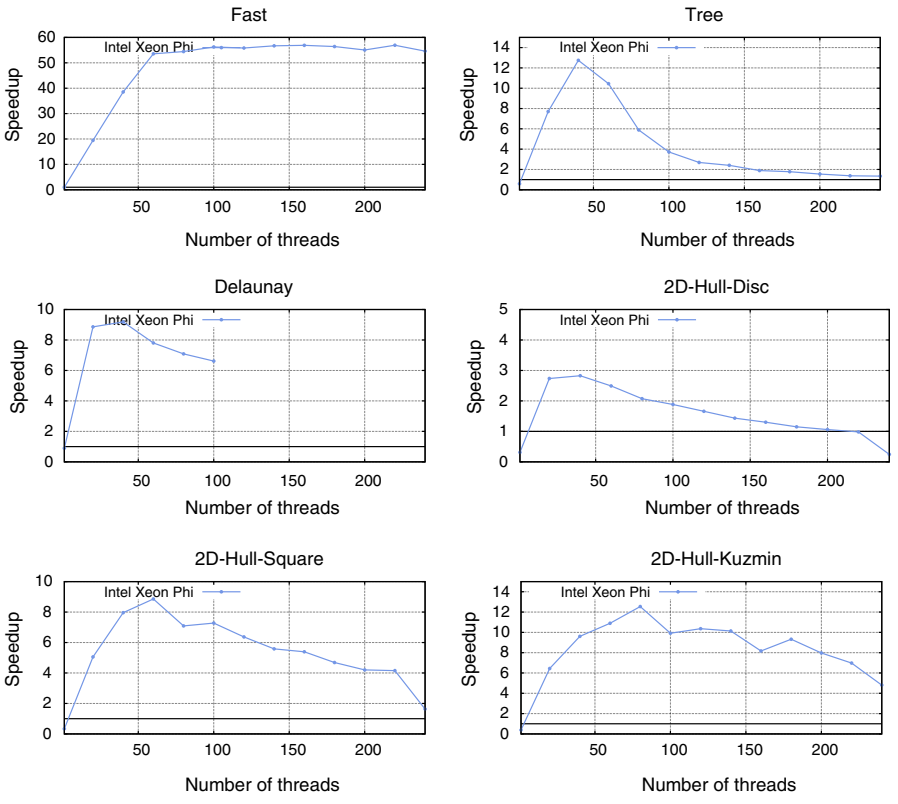

Fig. 4 Speedups by number of processors for each benchmark tested on the Intel Xeon Phi coprocessor

Xeon Phi coprocessors can manage up to 244 threads. However, due to the fact that threads of each core are not independent, from 61 threads on (there are in total 61 cores) most of these cores are of no use to execute speculative threads that follow their own execution path.

In conclusion, we have found that the particular architecture of the Xeon Phi, with threads working synchronously in each core, is not particularly suitable for software-based speculative execution.

5.3 Absolute Performance

Although the Xeon Phi presents a better scalability when comparing with a conventional, shared-memory system, when considering absolute times, the picture is very different. Figure 5 shows the absolute times required to run each benchmark in Heracles and in the Xeon Phi installed in Chimera. The analysis of this figure leads to two conclusions. First, the use of the Xeon Phi to execute these benchmarks in parallel reduces the execution time needed by a single, high-end processor for the Fast, Tree, and 2D-hull benchmarks, using the Square and the Kuzmin input sets. On the

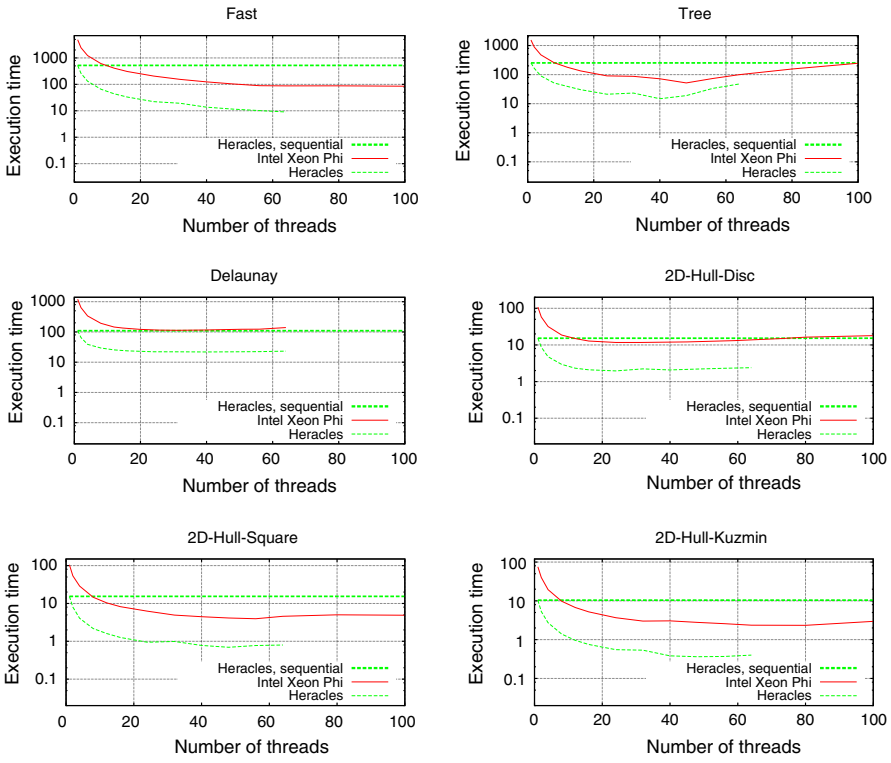

Fig. 5 Execution time in seconds with respect to the number of threads for each benchmark. The sequential time obtained with a single Xeon processor in Heracles is also shown

Table 1 Comparison of the time in seconds required to execute the benchmarks tested in both the Heracles, the shared memory system, and in the Xeon Phi coprocessor of Chimera

	32 threads			64 threads		
Application	(A) Xeon Phi	(B) Heracles	(A)/(B)	(A) Xeon Phi	(B) Heracles	(A)/(B)
FAST	154.45	19.28	8.01	87.68	9.03	9.71
2D-Hull, Disc	11.71	2.22	5.27	13.81	2.39	5.77
2D-Hull, Square	4.93	0.99	4.98	4.58	0.80	5.75
2D-Hull, Kuzmin	3.01	0.54	5.62	2.36	0.40	5.90
Delaunay	114.08	22.04	5.18	139.50	23.24	6.00
TREE	87.30	23.18	3.77	99.33	47.49	2.09

contrary, Delaunay and 2D-hull using the circle input set does not benefit at all from this architecture.

The second conclusion is that when comparing the absolute times obtained with the same number of threads in both architectures, we can see that the shared-memory architecture of Heracles allows to obtain execution times that are roughly an order of magnitude better than those produced by the Xeon Phi. The reasons are not only the higher clock speed of AMD processors, but their more advanced architecture, with out-of-order execution and branch prediction, compared with the in-order execution of the Pentium-based Xeon Phi computing units, that stall the execution in the case of a cache miss. The much more powerful architecture of AMD processors compensates the performance losses derived from the memory organization in the shared-memory system with respect with the one offered by the Xeon Phi, that leads to a better scalability as we saw in previous sections. Regarding the influence of the compiler chosen, in theory the choice of the Intel compiler might help the Xeon Phi platform to obtain better results. However, as long as the benchmarks are irregular, integer-based applications, the vectorization capabilities of the Intel compiler are of limited use in this case.

Table 1 summarizes the execution times for 32 and 64 threads in both architectures, with the corresponding relationship. As can be seen, relative speedups obtained by Heracles range from 2.09× for TREE with 64 processors, to 9.71× for FAST with 64 processors.

Despite the poor performance delivered, we consider that the Xeon Phi coprocessor may still help in the speculative execution of loops thanks to their comparatively big number of threads. Our future work include the combination of software-based TLS techniques with other solutions, such as value prediction, or the use of helper threads.

6 Related Work

To the best of our knowledge, this is the first research that tests TLS with Xeon Phi coprocessors. We will briefly review some related TLS approaches, and other studies that measure Xeon Phi capabilities.

6.1 TLS Approaches

Several researches have been centered on the parallelization of loops with cross-iteration dependences through thread-level speculation (TLS) techniques. Some of them have been implemented in hardware (e.g. [24,28,40], through the design of specific chips, or the addition of some functionalities. But there are also several software approaches that support the mentioned parallelism with no architectural changes [8,22,25,42]. In this work, we will describe some of the software approaches, and propose a number of possible hardware additions which might improve the performance of TLS on the Intel Xeon Phi coprocessors.

6.1.1 Software Branch

One of these software approaches is the work of Tian, Feng, Nagarajan and Gupta in [42], where they proposed the Copy-or-Discard (CorD) execution model, in which the execution of parallel threads are separately managed by the non-speculative one. Speculative threads read values of the non-speculative thread and perform their computation, after that, speculative threads are committed in order. After that results are checked by non-speculative thread in order to preserve semantics of sequential order. Commit operation is performed by non-speculative thread through the Copy or Discard mechanism that checks whether results are correct to be copied to the non-speculative data, or discarded with no cost otherwise. However, CorD approach did not support those applications whose speculative variables were dynamically allocated, so in [41] Tian, Feng and Gupta developed mechanisms that enable their solution to do it.

Cintra and Llanos [8,9] developed another scheme mainly based on an aggressive sliding window, with checks for data dependence violations on speculative stores that reduced synchronization constraints, and with fine-tuned data structures.

Kulkarni et al. [25,26], introduced *Galois*, a system to support complex pointer-based sets of elements in optimistic parallelism. They were centered on parallelize applications with complex structures as linked lists, graphs, trees, etc.

Oancea et al. [31] described their own TLS approach called *SpLIP*, centered on decreasing overheads of speculative operations of previous approaches. They implemented non-locking operations where possible, and used a hash function to improve location of version copies. Their hash is based on mapping adjacent zones of the array that stored speculative values in a single place. A similar approach to *SpLIP* [31], called *MiniTLS*, was developed by Yiapanis et *al.* [45].

Jimborean et al. [21] introduced a TLS framework specially designed to speculatively execute nested loops. To do so, authors used features of polyhedral model to dynamically transform code in a more optimized version that led to higher speedups. Framework consisted on dividing execution in two parts, one to generate some skeletons, and other one that selected the optimized code at runtime.

Most of the reviewed approaches might be useful to test the performance of the Xeon Phi. As we saw in previous section, we have used the ATLaS framework [4,15]. Our results suggest that the use of other software-only TLS solutions may lead to similar conclusions.

6.1.2 Hardware Improvements to Benefit Software TLS

We will now explore some enhancements which might possibly improve the performance of TLS on Intel Xeon Phi coprocessors. We will center our discussion on applying ideas belonging to the classical hardware approaches to manage speculation in multicore processors. Therefore, the implementation of these ideas would need changes in the Xeon Phi architecture.

Sohi et al. [40] developed the multiscalar processor, where cores were interconnected through a ring, an approach also followed in the speculative multithreaded processor [28]. For these systems, hardware modules developed to store intermediate versions of variables were also proposed, such as ARB [17] or SVC [19]. As long as the ring interconnection mechanism is also present in the Xeon Phi coprocessors', the application of their mechanisms to handle dependences in hardware might decrease software overheads.

Another possible improvement might be the addition of a new cache, based on the Trace cache [37]. This proposal stored traces (dynamic sequences of instructions stored in the hardware) at runtime, and instructions were executed in parallel, while dependences were speculated with the use of predictors.

A different approach like the used in the I-ACOMA architecture [24] may work as well. They used a binary annotator that added some notes into executable files to detect possible dependences, that were managed at runtime with a special module called memory disambiguation table. Another source of ideas for improvements is the threaded multi-path execution [44] approach, that was focused on prediction techniques. This proposal executed all possible branches of a loop, whilst there were enough resources, a situation that is likely to occur in Xeon Phi coprocessors.

6.2 Studies Related to the Xeon Phi Coprocessor

The Xeon Phi coprocessor is being extensively studied. Some papers have developed extensions to offloaded regions. For example, COSMIC [6] is a middleware integrated in the subjacent software that tries to ease and improve the performance of multiprocessing in Xeon Phi coprocessors. This work aimed to reduce imbalance and overheads through the management of resources. It handled offload regions and takes care of the request of coprocessors, cores and memory. Snapify [36] tried to reduce failure rates of Xeon Phi coprocessors. The underlying idea was taking snapshots during execution (saving the state of applications) and if an error was produced, the execution was restored to a correct, saved state, instead of being restarted.

Some essays are focused in the implementation of existing algorithms into coprocessors. For example, [33] developed a multi-node 1D FFT implementation on coprocessors; [27] implemented a sparse matrix-vector multiplication; and [32] developed a SQL engine that benefited from the inherent parallelism related to Xeon Phi coprocessors.

Furthermore, as it is the case, there are many other papers centered on the measurement of the performance obtained from a Xeon Phi. [38] was one of the first papers that used Intel Xeon Phi coprocessor (that was called Intel Knights Ferry) to

evaluate the performance of scientific applications. Later, Cramer et al. [11] evaluated the behavior of some OpenMP benchmarks in a Xeon Phi coprocessor. They affirmed that common OpenMP codes could be easily migrated to Intel Xeon Phi, gaining more parallel performance without adding overheads. This study was enhanced in [39]. [16] also tested the Xeon Phi through the development of some microbenchmarks.

7 Conclusions

In this work we have evaluated the behavior of the Xeon Phi coprocessor in the context of software-only, thread-level speculation (TLS), a parallel technique that optimistically executes in parallel sequential codes without a prior dependence analysis. Intel Xeon Phi coprocessors are one of the state-of-the-art architecture that aims to execute parallel codes. Our experimental results show that the particular memory architecture of the Xeon Phi leads to better scalability with regards to speculative execution, with better relative speedups than those obtained using a conventional, shared-memory architecture. However, the relatively low computing power of its computational units when specific vectorization and SIMD instructions are not exploited, indicates that further development of new specific techniques for this platform is needed to make it competitive for the application of speculative parallelization comparing with high-end processors or conventional shared-memory systems. This situation is likely to change with the arrival of the new generation of Xeon-Phi platforms, that incorporates more competitive processors. We plan to extend this work to this new generation, with a more detailed profile analysis, as soon as it becomes available.

Although the use of a Xeon Phi coprocessor to execute software-based, TLS codes is not competitive, the Xeon Phi architecture might be useful when combining TLS solutions with other existing techniques such as value prediction or helper threads. In this way, some of the available threads could be used to help TLS execution, reducing dependence violations and thus improving performance.

Acknowledgements This research is partly supported by the Castilla-Leon Regional Government (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).

References

1. AMD Opteron^{TM} 6300 Series processor - quick reference guide. https://www.amd.com/Documents/Opteron_6300_QRG.pdf. Accessed June 2015
2. Intel ® Xeon Phi^{TM} product family: Product brief. https://www-ssl.intel.com/content/dam/www/public/us/en/documents/product-briefs/high-performance-xeon-phi-coprocessor-brief.pdf. Accessed June 2015
3. Intel ® Xeon Phi^{TM} coprocessor instruction set architecture reference manual. https://software.intel.com/sites/default/files/forum/278102/327364001en.pdf. Accessed June 2015
4. Aldea, S., Estebanez, A., Llanos, D., Gonzalez-Escribano, A.: An OpenMP extension that supports thread-level speculation. IEEE Trans. Parallel Distrib. Syst. **PP**(99), 1–1 (2015). doi:10.1109/TPDS.2015.2393870
5. Barnes, J.E.: TREE. Institute for Astronomy. University of Hawaii (1997). ftp://hubble.ifa.hawaii.edu/pub/barnes/treecode/

6. Cadambi, S., Coviello, G., Li, C.H., Phull, R., Rao, K., Sankaradass, M., Chakradhar, S.: Cosmic: middleware for high performance and reliable multiprocessing on Xeon Phi coprocessors. In: Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing, HPDC '13, pp. 215–226. ACM, New York (2013). doi:10.1145/2462902.2462921
7. Cai, P., Cai, Y., Chandrasekaran, I., Zheng, J.: A GPU-enabled parallel genetic algorithm for path planning of robotic operators. In: Cai, Y., See, S. (eds.) GPU Comput. Appl., pp. 1–13. Springer, Singapore (2015). doi:10.1007/978-981-287-134-3_1
8. Cintra, M., Llanos, D.R.: Toward efficient and robust software speculative parallelization on multiprocessors. In: Proceedings of the SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) (2003)
9. Cintra, M., Llanos, D.R.: Design space exploration of a software speculative parallelization scheme. IEEE Trans. Parallel Distrib. Syst. **16**(6), 562–576 (2005)
10. Clarkson, K.L., Mehlhorn, K., Seidel, R.: Four results on randomized incremental constructions. Comput. Geom. Theory Appl. **3**(4), 185–212 (1993)
11. Cramer, T., Schmidl, D., Klemm, M., an Mey, D.: OpenMP programming on Intel Xeon Phi coprocessors: An early performance comparison. In: Proceedings of the Many-core Applications Research Community (MARC) Symposium (2012)
12. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998). doi:10.1109/99.660313
13. Devroye, L., Mücke, E.P., Zhu, B.: A note on point location in Delaunay triangulations of random points. Algorithmica **22**, 477–482 (1998)
14. Dou, J., Cintra, M.: Compiler estimation of load imbalance overhead in speculative parallelization. In: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, PACT '04. IEEE Computer Society, Washington, DC (2004)
15. Estebanez, A., Llanos, D., Gonzalez-Escribano, A.: New data structures to handle speculative parallelization at runtime. Int. J. Parallel Program. 1–20 (2015). doi:10.1007/s10766-014-0347-0
16. Fang, J., Sips, H., Zhang, L., Xu, C., Che, Y., Varbanescu, A.L.: Test-driving Intel Xeon Phi. In: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering, ICPE '14, pp. 137–148. ACM, New York (2014). doi:10.1145/2568088.2576799
17. Franklin, M., Sohi, G.S.: ARB: a hardware mechanism for dynamic reordering of memory references. IEEE Trans. Comput. **45**(5), 552–571 (1996). doi:10.1109/12.509907
18. Gao, L., Li, L., Xue, J., Yew, P.C.: SEED: a statically-greedy and dynamically-adaptive approach for speculative loop execution. IEEE Trans. Comput. **62**(5), 1004–1016 (2013)
19. Gopal, S., Vijaykumar, T.N., Smith, J., Sohi, G.: Speculative versioning cache. In: High-Performance Computer Architecture, 1998. Proceedings, 1998 Fourth International Symposium on, pp. 195–205 (1998). doi:10.1109/HPCA.1998.650559
20. Jeffers, J., Reinders, J.: Intel Xeon Phi Coprocessor High-Performance Programming. Newnes, Boston (2013)
21. Jimborean, A., Clauss, P., Dollinger, J.F., Loechner, V., Martinez Caamao, J.: Dynamic and speculative polyhedral parallelization using compiler-generated skeletons. Int. J. Parallel Program. **42**(4), 529–545 (2014)
22. Kelsey, K., Bai, T., Ding, C., Zhang, C.: Fast track: a software system for speculative program optimization. In: Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '09, pp. 157–168. IEEE Computer Society, Washington, DC (2009). doi:10.1109/CGO.2009.18
23. Khronos: Open Computing Language (OpenCL) (2010). http://www.khronos.org/opencl/, Accessed 2 Dec 2013
24. Krishnan, V., Torrellas, J.: A chip-multiprocessor architecture with speculative multithreading. IEEE Trans. Comput. **48**(9), 866–880 (1999)
25. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L.P.: Optimistic parallelism requires abstractions. In: PLDI 2007 Proceedings. ACM (2007)
26. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L.P.: Optimistic parallelism requires abstractions. Commun. ACM **52**(9), 89–97 (2009)
27. Liu, X., Smelyanskiy, M., Chow, E., Dubey, P.: Efficient sparse matrix-vector multiplication on x86-based many-core processors. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS '13, pp. 273–282. ACM, New York (2013). doi:10.1145/2464996.2465013

28. Marcuello, P., Gonzalez, A., Tubella, J.: Speculative multithreaded processors. In: Proceedings of the 12th International Conference on Supercomputing, ICS '98. ACM, New York (1998)
29. Mücke, E.P., Saias, I., Zhu, B.: Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations. In: SoCG '96 Proceedings, pp. 274–283 (1996)
30. NVIDIA: NVIDIA CUDA Architecture Introduction and Overview Version 1.1 (2009)
31. Oancea, C.E., Mycroft, A., Harris, T.: A lightweight in-place implementation for software thread-level speculation. In: Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures, SPAA '09. ACM, New York (2009)
32. Olsen, S., Romoser, B., Zong, Z.: SQLPhi: A SQL-based database engine for Intel Xeon Phi coprocessors. In: Proceedings of the 2014 International Conference on Big Data Science and Computing, BigDataScience '14, pp. 17:1–17:6. ACM, New York (2014). doi:10.1145/2640087.2644172
33. Park, J., Bikshandi, G., Vaidyanathan, K., Tang, P.T.P., Dubey, P., Kim, D.: Tera-scale 1D FFT with low-communication algorithm and Intel Xeon Phi coprocessors. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13, pp. 34:1–34:12. ACM, New York (2013). doi:10.1145/2503210.2503242
34. Raman, E., Vahharajani, N., Rangan, R., August, D.I.: Spice: speculative parallel iteration chunk execution. In: Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '08. ACM, New York (2008)
35. Rauchwerger, L., Padua, D.: The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization (1995). doi:10.1145/207110.207148
36. Rezaei, A., Coviello, G., Li, C.H., Chakradhar, S., Mueller, F.: Snapify: capturing snapshots of offload applications on Xeon Phi manycore processors. In: Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC '14, pp. 1–12. ACM, New York (2014). doi:10.1145/2600212.2600215
37. Rotenberg, E., Bennett, S., Smith, J.E.: Trace cache: a low latency approach to high bandwidth instruction fetching. In: Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture. MICRO 29, pp. 24–35. IEEE Computer Society, Washington, DC (1996)
38. Satish, N., Kim, C., Chhugani, J., Saito, H., Krishnaiyer, R., Smelyanskiy, M., Girkar, M., Dubey, P.: Can traditional programming bridge the ninja performance gap for parallel computing applications? In: Proceedings of the 39th Annual International Symposium on Computer Architecture, ISCA '12, pp. 440–451. IEEE Computer Society, Washington, DC (2012). http://dl.acm.org/citation.cfm?id=2337159.2337210
39. Schmidl, D., Cramer, T., Wienke, S., Terboven, C., Mller, M.: Assessing the performance of OpenMP programs on the Intel Xeon Phi. In: Wolf, F., Mohr, B., an Mey, D. (eds.) Euro-Par 2013 Parallel Processing, *Lecture Notes in Computer Science*, vol. 8097, pp. 547–558. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40047-6_56
40. Sohi, G.S., Breach, S.E., Vijaykumar, T.N.: Multiscalar processors. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture, ISCA '95, pp. 414–425. ACM, New York (1995). doi:10.1145/223982.224451
41. Tian, C., Feng, M., Gupta, R.: Supporting speculative parallelization in the presence of dynamic data structures. In: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10. ACM, New York (2010)
42. Tian, C., Feng, M., Nagarajan, V., Gupta, R.: Copy or discard execution model for speculative parallelization on multicores. In: Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '41. Washington, DC (2008)
43. Walker, D.W.: The design of a standard message passing interface for distributed memory concurrent computers. Parallel Comput. **20**(4), 657–673 (1994). http://portal.acm.org/citation.cfm?id=180103
44. Wallace, S., Calder, B., Tullsen, D.M.: Threaded multiple path execution. In: Proceedings of the 25th Annual International Symposium on Computer Architecture, ISCA '98, pp. 238–249. IEEE Computer Society, Washington, DC (1998). doi:10.1145/279358.279392
45. Yiapanis, P., Rosas-Ham, D., Brown, G., Luján, M.: Optimizing software runtime systems for speculative parallelization. ACM Trans. Archit. Code Optim. **9**(4), 39:1–39:27 (2013)
46. Zhao, Z., Wu, B., Shen, X.: Speculative parallelization needs rigor: probabilistic analysis for optimal speculation of finite-state machine applications. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, PACT '12. New York (2012)