7th International Symposium on
High-Level Parallel Programming and Applications (HLPP 2014)
Amsterdam, Netherlands
July 3-4, 2014

New Data Structures to Handle Speculative Parallelization at Runtime

Alvaro Estebanez · Diego R. Llanos · Arturo Gonzalez-Escribano

Abstract Software-based, thread-level speculation (TLS) is a software technique that optimistically executes in parallel loops whose fully-parallel semantics can not be guaranteed at compile time. Modern TLS libraries allow to handle arbitrary data structures speculatively. This desired feature comes at the high cost of local store and/or remote recovery times: The easier the local store, the harder the remote recovery. Unfortunately, both times are on the critical path of any TLS system. In this paper we propose a solution that performs local store in constant time, while recover values in a time that is in the order of T, being T the number of threads. As we will see, this solution, together with some additional improvements, makes the difference between slowdowns and noticeable speedups in the speculative parallelization of non-synthetic, pointer-based applications on a real system. Our experimental results show a gain of 3.58× to 28× with respect to the baseline system, and a relative efficiency of up to, on average, 65% with respect to a TLS implementation specifically tailored to the benchmarks used.

Keywords thread-level speculation · speculative parallelism · memory improvements

1 Introduction

Thread-Level Speculation (TLS) [3,29,30,34], also called Speculative Parallelization (SP) [8,10,14,37] or Optimistic Parallelism [16–21] tries to extract parallelism of loops that can not be considered fully parallel at compile time.

A. Estebanez · D.R. Llanos · A. Gonzalez-Escribano
Departamento de Informatica, Universidad de Valladolid, Valladolid, Spain 47011
E-mail: alvaro@infor.uva.es

D.R. Llanos
E-mail: diego@infor.uva.es

A. Gonzalez-Escribano
E-mail: arturo@infor.uva.es

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 proceed. 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 that are not private nor shared are labeled at compile time as "speculative". 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.

The most common solution to maintain speculative data is to allow each thread to keep a version copy of all the speculative variables that have been locally accessed. Once a thread finishes the execution of its chunk of iterations, all changes in the speculative data are committed to main memory.

The biggest challenge in software-based TLS is how to reduce the time needed to (a) get the most up-to-date value when reading speculative data, and (b) to search for a possible dependence violation when a thread writes on a speculative variable. Note that both operations imply traversing all the version copies maintained by other threads. In the first case, the search for an up-to-date value implies to traverse all the data being kept by all predecessor threads (that is, threads being executing earlier chunks of iterations). In the second case, the search implies to traverse all the speculative data being maintained by all successors.

Access to predecessor and successor copies of the data are in the critical path of any TLS system. The problem is even more difficult to solve if the TLS library allows to speculate over dynamic structures and/or pointer-based references.

Among all software-based TLS approaches proposed in the literature, few of them are capable of speculatively handling dynamic data structures [20, 34]. Considering the difficulty of the problem to solve, it is not strange that the solutions proposed rely on abstract approaches that are difficult to both understand and implement.

The contribution of this paper is to show a new solution to traverse speculative data in a software-based TLS library. We will describe how to dramatically decrease the number of memory accesses when searching for predecessor and successor versions of speculative data, while keeping the cost of local data storage in $O(1)$. Our experimental results with well-known benchmarks on a real system show that these optimizations lead to significant reductions in the number of accesses needed (by a factor of three orders of magnitude) comparing with a competitive baseline implementation that lacks this feature. We also propose additional solutions to further reduce the memory allocation calls,

needed to dynamically add new variables to the speculative structures that should be managed at runtime. The combined effect of all these improvements is an impressive increment in the speedups obtained.

The rest of this paper is structured as follows: Section 2 puts this work in perspective with the existent solutions in this field. Section 3 describes the baseline solution developed in a previous research. Section 4 describes both the experimental environment and the benchmarks used to evaluate the baseline solution and our new proposals. Section 5 evaluates the costs of speculative reads and writes, from a theoretical and experimental points of view. Section 6 addresses the improvements applied to the library in order to improve its performance. Section 7 describes two additional improvements that leads to an ever faster speculative parallel execution. Section 8 shows some experimental results in terms of performance measured in a real system. Finally, Section 9 concludes this paper.

2 Related work

Memory management is a critical field in the context of speculative parallelization. As we will see in this paper, an adequate handling of speculative variables makes a huge difference in terms of performance. We will first review some efforts in this field, and later we will concentrate on the problem of ensuring fast accesses to version data.

2.1 TLS approaches

Several researches have been centered in the parallelization of loops with cross-iteration dependences through thread-level speculation (TLS) techniques. Some of them have been implemented in hardware [5, 7, 11, 15, 24, 31, 32], 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 [3, 14, 16, 18, 20, 23, 28, 30, 34]. One of this software approaches is the work of Tian, Feng, Nagarajan and Gupta in [34, 35], 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 [33] Tian, Feng and Gupta developed mechanisms that enable their solution to do it.

Cintra and Llanos [3,4] contributed 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.* in the work described in [20,21], 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.

In particular, these approaches suffered from some specific bottlenecks because a dynamic structure could change their size during the execution, and usually they are bigger than the static structures, so traverse them could be very inefficient.

2.2 The problem of traversing version copies

Earlier software-based TLS libraries such as [3] only allowed speculative accesses to static data structures such as vectors. In this way, if a given thread wanted to get the most up-to-date value for the i-th element of a speculative vector, it simply looked for it in the i-th position of the version copy of each predecessor. Writes were handled in the same way, looking for the i-th position of each successor's version copy of the speculative data. It is easy to see that, with T threads, the complexity of this search is in $O(T)$, an affordable value taking into account that T is usually in the order of tenths of processors.

Modern TLS libraries allow to speculatively access not only static vectors, but arbitrary memory locations. Usually, this data is kept with no particular order, an attractive solution since the cost of inserting a new element to the structure is exactly in $O(1)$. The problem with this solution is that the search for a previous value (due to a read operation) and the search for potential violations (due to a write) imply traversing all version data kept by all predecessors and successors, respectively. If we consider T threads and M data elements in each version copy, this lead to a complexity that is in $O(T \times M)$, with M potentially in the order of millions of elements. The solution of storing values using an ordered data structure may seem reasonable, but this comes at the cost of a higher storage time, that is also undesirable. As we will see, our solution keeps the storage cost in $O(1)$ while the traversing cost is in $O(T \times \frac{M}{H})$, with H an arbitrarily large constant. In practice, this solution leads to a traversing cost that is just in $O(T)$.

Ceze *et al.* [2] addressed the problem of the complexity of the basics operations involved in TLS processes. To do this they used a kind of hash encoder, called signature, that manages the addresses accessed by each speculative thread. Each signature is a set of addresses and allowed to treat several addresses as if they were a single one. They enhanced the architecture with some hardware mechanism that could efficiently operate with this hashed information. The main differences with our system are that their's is hardware-based

and the hash is used to perform operations to a group of addresses instead traversing their data.

Kulkarni et *al.* [19] introduced some improvements that increased the efficiency of *Galois*. They implemented a method to perform data partitioning in the data in a way that all elements of a set where mapped to an abstract domain, and then, transformed again to physical cores. Mendez-Lojo et *al.* [26] also described three techniques to optimize irregular applications. The first one is modifying codes in a way that all read operations are done before any write operation. The second one, based on the calculation of dependences before the execution. The last one, is used for those algorithms whose bottlenecks are located in the accesses to data sets and it is based on removing the correspondence between iteration and activities. Unlike our solution, all these optimizations are referred to algorithms.

Tian et *al.* [33] addressed the problems related to storage and location of intermediate values in CorD with the use of a mapping table that translate addresses between speculative and non-speculative threads. In that work, authors affirmed that using a hash function was inefficient, with a $6\times$ slowdown with respect to other alternatives due to the use of a complex hash function. Our work shows that the use of a simpler hash function leads to impressive speedups in several applications.

Mehrara et *al.* [25] described *STMLite*, a software transactional memory model modified to support speculative parallelization. It was specially designed to reduce overheads of accesses to variables logs in transactions using a thread that managed the execution. Management of addresses was handled with the use of hash-based solutions. More software transactional model systems have used hashes to improve their performance, i.e., Harris et *al.* [12] used hashes to remove duplicates in an undo log.

Oancea et *al.* [27] described their own TLS approach called *SpLIP*, centered on decreasing overheads of speculative operations of previous approaches. They implemented non-locking operations where was 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. As we will see, our solution is not based on joining near addresses, since we do not need to group speculative variables.

A similar approach to *SpLIP*[27], called *MiniTLS*, was developed by Yiapanis et *al.* [36]. They introduced a new structure that optimized memory overheads of classical approaches based on the idea of mapping every user-accessed address into an array of integers using a hash function.

Jimborean et *al.* [13] 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.

3 Description of the baseline solution

Software speculative schemes should allocate some additional memory in order to hold the information related to speculative executions. The use of this data is mandatory to enable recovery operations that could arise in an optimistic execution. In this context, memory needed could be allocated dynamically, or statically, and the use of an approach instead of the other is a critical decision that directly influences in the overall memory used in a program.

Our entire research framework relies on a first, pointer-based version of a software-based TLS library that strictly follows the principles of the library developed by Cintra and Llanos [3,4]. That works established the foundations of the correctness of the speculative execution of sequential applications. Their approach only allowed to speculate on variables encapsulated inside a vector. The use of this solution required threads to allocate memory for the entire vector, even if many positions of it were not used during the execution of the assigned chunk of iterations. So, in the case that M was the speculative variables used in a problem executed with T threads, and each variable need a byte, variables require $M \times (T+1)$ bytes to be stored, because an additional space is required to save the persistent copy. Moreover, using vectors required that all the variables used had to share the same type.

The version that we use as a baseline in this work has been originally presented in [9]. This baseline version 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, referencing these variables either by name or by address. As we will see, although this library effectively removes many constrains of Cintra and Llanos' solution, the strict adherence to the original architecture leads to unacceptable costs for speculative reads and writes. In this section we will briefly show the architecture of our library, since it will be used as the baseline to test some improvements proposed in this paper.

3.1 Data structures

The data structures needed by the baseline speculative library are depicted in Fig. 1. A matrix with W window slots (four in the figure) implements a sliding window that manages the runtime of the library. Each slot is responsible to manage the speculative execution of a particular set of iterations. The slots assigned to the non-speculative and the most-speculative threads are indicated by two variables, `non-spec` and `most-spec`. 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.

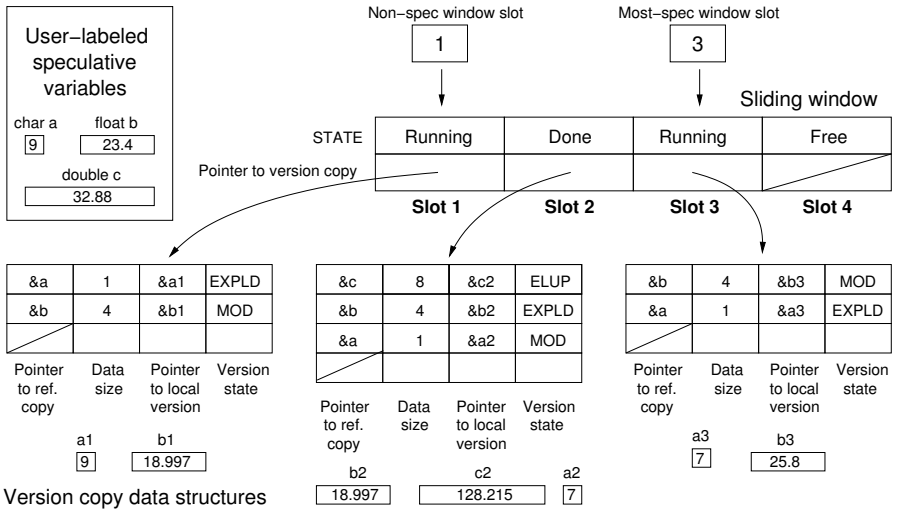

Fig. 1 Data structures of our new speculative library.

An example of the execution of a loop is also depicted in Fig. 1. The loop has been divided into three chunks of iterations, and it will be executed in parallel using three threads. 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 the corresponding slot to work in. This allows to maintain an order relationship among the chunks being executed.

In the depicted example, thread working in slot 1 is executing the non-speculative chunk of iterations (as indicated by its RUNNING state); the following chunk has been already executed and its data has been left there to be committed after the non-spec chunk finishes (since it is in DONE state), while the last one, the most-speculative chunk launched so far, is also RUNNING. In other words, the thread in charge of the second chunk has already finished, while the non-spec and most-spec threads are working. If more chunks were pending, the freed thread would be assigned the following chunk, starting its execution in slot 4. Slot 2 can not be re-used yet, because the execution of chunk 2 left changes to speculative variables that are yet to be committed. As we will see in Sect. 3.3, when the non-speculative thread working in slot 1 finishes, it will commit its results and the results stored in all subsequent DONE slots, since commits should be carried out in order. After that, in our example, the non-spec pointer will be advanced to slot 3 to reflect the new situation.

In addition to its STATE, each slot points to a data structure that holds the version copies of the data being speculatively accessed. Fig. 1 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

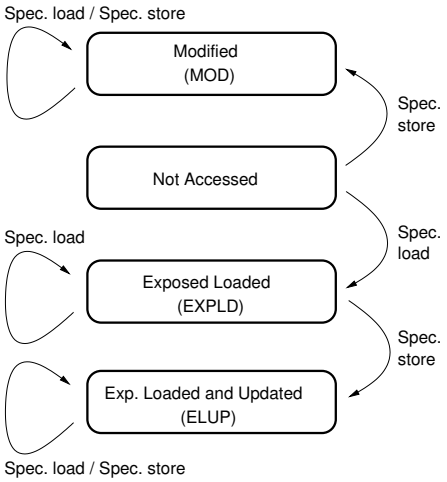

Fig. 2 State transition diagram for speculative data.

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. The transition diagram for these states is shown in Fig. 2.

Fig. 1 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.

3.2 Speculative loads and stores

The interface of our implementation of specload() is as follows:

```
specload(VOID* addr, UINT size, UINT chunk_number, VOID* value)
```

The first parameter is the address of the speculative variable; the second one is the size of the variable; the third one is the number of the chunk being executed (needed to infer the slot being used); and the fourth one is a pointer to a place to store the datum requested.

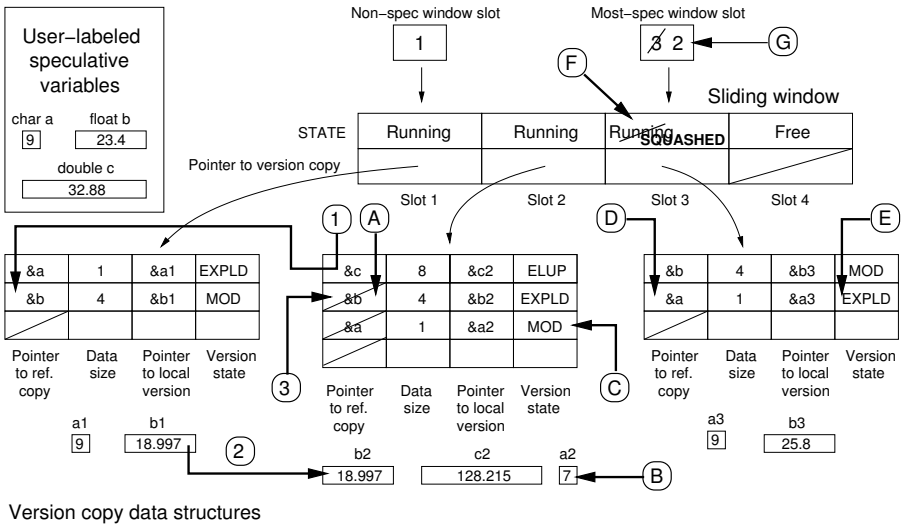

Fig. 3 Steps of a speculative load (1..3) and speculative store (A..G).

Recall that specload() should return the most up-to-date value available for the speculative variable. Fig. 3 shows how speculative load works. Suppose that the thread working in slot 2 has only accessed variable c so far, and then it calls specload(&b, sizeof(b), 2, &value) to obtain a value for b. The thread working in slot 2 scans from its version copy data structure to its predecessors' until the value is found (point 1). Otherwise, if value has not been used before, it is obtained from reference copy. In the Fig. 3 the thread working in slot 1 has used b, so the thread that called specload() copies the value to its own local copy (point 2), and add a row to its version copy data structure. Note that the process described in the point 1 of Fig. 3, that is, searching for a copy of a variable is a sequential process that origins big bottlenecks.

The interface of specstore() is similar as specload()'s, but in this case the last parameter is a pointer to the value to be stored. Recall that specstore() should not only store the new value, but also check whether a successor has consumed an outdated value for it. Fig. 3 shows the sequence of events related to a speculative store. Suppose that the thread working in slot 2 executes specstore(&a, sizeof(a), 2, &temp), where temp holds the value 7. Thread working in slot 2 searches for a local version copy of a in its structure (point A). If it was found, its local value would be updated, but in this case, a new row is added with the address of a, its size, the address of the new local version (point B), and its state (point C). Then, the thread that performed the call should check whether any successor thread has consumed an outdated value of a (point D). In this case, the thread working in slot 3 has loaded this value (point E), so, it should be squashed (points F and G). Note that the process described in the point D of Fig. 3, that is, searching for copies of outdated variables is a sequential process that origins big bottlenecks.

3.3 Partial commit operation

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 to be committed by the non-spec thread.

Suppose that we are in the situation depicted in Fig. 1, and the non-spec thread working in slot 1 finishes. As long as it is the non-spec one, it will scan its data structure for variables in ELUP or MOD state. In our example, b has been modified, so it copies the content of b1 into b. After committing the version copy data structure associated to slot 1, it changes its state to FREE and advances the non-spec pointer to 2. As long as slot 2 is marked as DONE, its data should be committed as well. In our example, data stored in c2 and a2 should be committed to the user-defined variables. After this, the state of the slot is also changed to FREE and the non-spec pointer is advanced as well. Thread working in slot 3 is still running: When it finishes, it will be in charge of committing its own data. These commit operations are carried out with the help of auxiliary data structures that store a list of elements in ELUP or MOD states (not shown in our examples), in order to avoid traversing the local copies entirely only to commit few data elements.

It is interesting to note 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

Before describing the improvements proposed, an in-depth evaluation of the baseline solution is needed. This section describes our experimental environment, including the benchmarks used and the target system. The following section will show the behaviour of the baseline solution in the execution of these benchmarks.

To test both the baseline TLS library and our improvements we have used the 2-dimensional Convex Hull problem (2D-Hull), and the Delaunay triangulation. Output results of these benchmarks deterministically depend on their input sets.

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.* [6] 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 been used the old enclosing polygon to process the points assigned to them. We have used

Application	Input set	Average violations (#)	Average violations (%)
2D-Hull	Kuzmin 10M	53.5	0.005%
2D-Hull	Square 10M	180.25	0.018%
2D-Hull	Disc 10M	705.37	0.071 %
Delaunay	1M	2998.25	0.299%
Delaunay	100K	1657.00	1.657 %

Table 1 Characteristics of the benchmarks considered.

three different input sets composed by 10M of points: Kuzmin, that processes a 2D Kuzmin distribution; Square, that processes a 2D uniform distribution located into a square, and Disc, with an uniform, 2D distribution located into a disc.

The Delaunay triangulation [22] applied to a two-dimensional set of points affirms that a network of triangles is a Delaunay triangulation if all the circumcircles of all the triangles of the network are empty, i.e., the circumcircle of each triangle of the network contains no other vertices that those three that define the triangle. This condition ensures that the interior angles of the triangles are as large as possible and the length of the sides of the triangles is minimal. We have used two different input sets, of 100K and 1M points.

Table 1 summarizes the main characteristics of both applications and their corresponding input sets. The average violations have been calculated executing the benchmarks in parallel with 2 to 16 threads in our target system. Note that the number of dependence violations that arise at runtime is not zero for any of them. These violations severely limit the speedup that can be obtained with any speculative, parallel execution.

Experiments were carried out on an Intel S7000FC4URE server, equipped with four quad-core Intel Xeon MPE7310 processors at 1.6GHz and 32GB of RAM. The system runs Ubuntu Linux operating system. All threads had exclusive access to the processors during the execution of the experiments, and we used wall-clock times in our measurements. We have used gcc 4.6.2 for all applications. Times shown in the following sections represent the time spent in the execution of the parallelized loop for each application. The time needed to read the input set and the time needed to output the results have not been taken into account.

5 Baseline cost of speculative reads and writes

One of the main advantages of the baseline speculative parallelization library is that each thread only allocates the memory needed to store local copies of the data being speculatively accessed. This design decision comes at the cost of longer times to find the most-up-to-date value in speculative loads, and longer times to detect dependence violations in speculative stores, since both operations should traverse all the values accessed by all the predecessors and successors, respectively. Being T the number of threads, and M the number of

		speculative load			speculative store		
App	Input set	TC	TS	TS/TC	TC	TS	TS/TC
2D-Hull	Disc	2.63×10^{8}	2.23×10^{10}	87.39	4.39×10^{4}	2.63×10^{7}	598.10
	Square	2.97×10^{8}	3.05×10^{10}	102.86	6.65×10^{3}	4.90×10^{6}	737.35
	Kuzmin	2.29×10^{8}	1.13×10^{10}	49.35	1.65×10^{3}	5.30×10^{5}	320.64
Delaunay	100K	1.14×10^{7}	3.77×10^{8}	33.18	5.06×10^{6}	1.69×10^{8}	33.44
	1M	1.47×10^{8}	8.17×10^{9}	55.42	5.28×10^{7}	3.29×10^{9}	62.32

Table 2 Profile of main functions with a single thread in the baseline TLS library.

		speculative load			speculative store		
App	Input set	TC	TS	TS/TC	TC	TS	TS/TC
2D-Hull	Disc	2.74×10^{8}	4.00×10^{10}	145.85	7.48×10^{4}	1.14×10^{8}	1526.99
	Square	3.17×10^{8}	3.55×10^{10}	111.86	2.40×10^{4}	3.30×10^{7}	1374.33
	Kuzmin	2.35×10^{8}	1.18×10^{10}	50.18	5.78×10^{3}	4.22×10^{6}	729.73
Delaunay	100K	1.14×10^{7}	1.42×10^{9}	124.83	5.07×10^{6}	7.51×10^{8}	148.09
	1M	1.47×10^{8}	3.24×10^{10}	219.89	5.28×10^{7}	1.31×10^{10}	248.69

Table 3 Profile of main functions with eight threads in the baseline TLS library.

data elements stored locally (that is potentially very high), the search is done in $O(T \times M)$. Therefore, the performance figures for the baseline library with this mechanism were severely limited. In fact, as we will see in Sect. 8, none of the speculative executions of the applications described in the previous section broke even in terms of speedup.

Tables 2 and 3 show the total calls to *specload()* and *specstore()* operations for each benchmark. In this table, TC (total calls) is the total number of calls to each function; TS (total searches) is the total number of accesses in order to complete the corresponding call (getting the most up-to-date value in *specload()*, and searching for potential dependence violations in *specstore()*). These numbers are average values obtained in three real executions in our target system.

Table 2 shows the values obtained when using the TLS library to execute each benchmark, but using just one thread. This means that all accesses counted in TS were to the local version data. As it can be seen, this number of accesses in speculative loads is high on average, from 33.18 to 102.86. The situation is much worse on speculative stores, with up to 737 accesses on average in order to detect a potential dependence violation. Compare these numbers with the single access needed by both reads and writes in a non-speculative execution of the same algorithm.

Table 3 shows the same values when we speculatively execute the benchmarks with eight threads. Both the costs of speculative loads and stores are roughly doubled. The situation becomes even worse as we increment the number of cooperative threads.

These figures show that the main bottleneck and the most severe scalability limitation come from the sequential traversing of version copies during speculative loads and stores operations. One way to speed up these searches is to switch to a different data structure to hold local version copies of data. Instead of using a single table per thread as version copy data structure, we

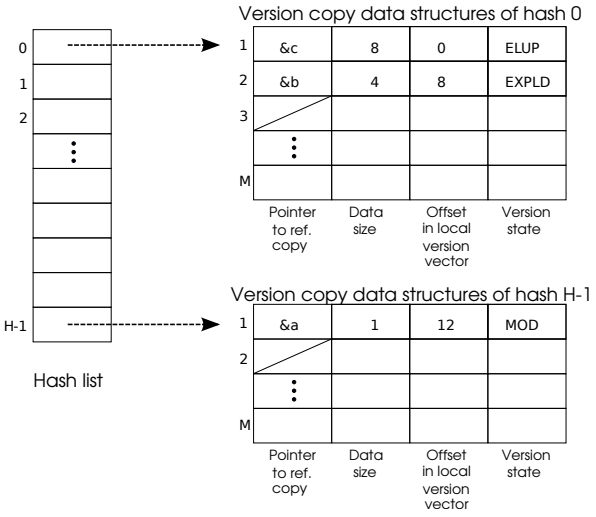

Fig. 4 Hash-based version copy data structures.

have developed an simple but extremely powerful alternative, using a hash function and H tables.

6 A hash-based solution

We have devised an extremely simple solution that is capable of reducing the number of accesses needed by a factor of one to three orders of magnitude, while keeping the storage cost of local versions in $O(1)$. In addition, the developed technique can be seen as a general solution that can be applied to most TLS systems, not only to our baseline implementation. The solution works at follows: At the beginning of each *specload()* and *specstore()* call, we perform an AND operation of the address of the datum to be processed with a mask composed by H 1s. Since many addresses are multiple of 4 or 8, the address is first shifted three positions to its right, to avoid biased hash values. The resulting value will be used as a hash value. Considering an address A, the operation is:

$$\text{hash} = (A >> 3) \wedge 00...0\overbrace{11...1}^{H}$$

Instead of having just one table to keep local values, each thread maintains H tables. The obtained hash is used to look into the H-th table of all predecessors and successors, effectively speeding up the search by an average factor of H without increasing the time needed to add a new row to the corresponding table, leading to $O(T \times \frac{M}{H})$ search times. Note that, while T is a relatively small number (typically up to 64 for current shared memory architectures),

		speculative load			speculative store		
App	Input set	TC	TS	TS/TC	TC	TS	TS/TC
2D-Hull	Disc	2.62×10^8	3.46×10^8	1.32	4.39×10^4	1.47×10^5	3.34
	Square	2.97×10^8	3.94×10^8	1.33	6.65×10^3	2.53×10^4	3.80
	Kuzmin	2.29×10^8	2.78×10^8	1.21	1.65×10^3	3.45×10^3	2.08
Delaunay	100K	1.14×10^7	1.74×10^7	1.53	5.06×10^6	8.18×10^6	1.62
	1M	1.47×10^8	2.35×10^8	1.59	5.28×10^7	9.02×10^7	1.71

Table 4 Profile of main functions with a single thread in the hash-based version of the library.

		speculative load			speculative store		
App	Input set	TC	TS	TS/TC	TC	TS	TS/TC
2D-Hull	Disc	2.74×10^8	4.47×10^8	1.63	7.24×10^4	4.55×10^5	6.29
	Square	3.19×10^8	4.38×10^8	1.37	2.28×10^4	1.42×10^5	6.23
	Kuzmin	2.34×10^8	2.85×10^8	1.22	7.18×10^3	2.16×10^4	3.01
Delaunay	100K	1.14×10^7	6.06×10^7	5.32	5.07×10^6	8.67×10^6	1.71
	1M	1.47×10^8	7.43×10^8	5.04	5.28×10^7	1.08×10^8	2.05

Table 5 Profile of main functions with eight threads in the hash-based version of the library.

H can be set as big as needed to compensate for higher values of M. Fig. 4 shows the new hash-based version copy structure.

Tables 4 and 5 show the total calls to *specload()* and *specstore()* operations, with the total accesses that each one of them needed to find the desired value, and the average accesses per call, in the new hash-based solution with one and eight processors respectively for each benchmark used. Compare the values in both tables for TS/TC columns (up to 6.29 accesses) with the much higher values (up to 1500) of the sequential searches shown in Tab. 2 and 3. Moreover, this solution comes at no cost, since the hash function is extremely easy to implement.

6.1 Memory consumption

New structures could seem to use more space than the previous ones, however, the memory used are similar in both schemes: Let us suppose that in the first approach each thread managed N rows to store intermediate values. If there were T threads, the cost to store them was $T \times O(N)$. On the new scheme, with H hash rows that contain M positions to store values, considering that $H \times M = N$ the cost to store them is $T \times O(H \times M)$. In the best case, if values were uniformly dispersed throughout the hash rows, $H \times M \approx N$. In the worst case, if a single hash stored all the N values, the cost of the new approach would be $(T-1) \times O(H \times M) + T \times O(N) \approx O(T \times N)$. So, required memory to store the new approach will be similar to the previous one.

7 Further improvements

7.1 Use of dedicated buffers

One of the problems detected was the excessive number of calls to the *malloc()* and *free()* functions. To better understand the reasons, we will use an example. Suppose that a thread executes a *specload()* or *specstore()* call. In both functions, the first task carried out by the thread is to search in its version copy matrix to check whether this address has been accessed by this thread. Suppose that the datum has not been used yet, so it should be added to the matrix. In this process of attaching the new datum to the matrix of the thread, we have to allocate some memory to store the local copy, so *malloc()* should be called (see step B of Fig. 3).

Once the thread has finishing the speculative execution of the chunk of iterations that has been assigned, it should free all its allocated memory. Therefore, *free()* should be repeatedly called to deallocate each single version copy of speculative data, in order to reuse the remaining data structures to handle the execution of a new chunk of iterations.

It is easy to see that these operations are costly because they are called very frequently. Moreover, the calls to *malloc()* are in the critical path of the speculative execution. A solution to avoid these calls is to implement a container for all the data used by each thread. Hence, a new, *Local Version Data* dynamic vector was added to each thread. These vectors need an initial call to the *malloc()* function to allocate them, and a single *free()* call to deallocate them once the parallel loop has been executed entirely. If the vector is full, an additional call to *realloc()* may be needed. This solution greatly improves the performance observed.

This new structure, however, leads to changes in the basic structures of the architecture. Initially we had an structure with four entries, where one of them was a pointer to the local copy of the datum. Instead, the new solution manages an offset for each datum. In this way, each datum will be stored in the Local Version Data vector at the position pointed by its offset. Note that, once stored, the datum will not be deleted until the entire vector containing speculative data is committed, so fragmentation will not occur.

Also, each slot of the sliding window requires an additional pointer to the first free position of its vector, to allow fast insertions. Hence, the sliding window is augmented with this additional datum. Fig. 5 shows a simple example that reflects our solution. Note that, after executing a chunk of iterations, it is not longer needed to *free()* this buffer, since it is enough to reset the index that points to the beginning of the free space on it.

Being T the number of threads, and M the number of data elements stored locally, the original solution came at a cost that was in $T \times O(M) \subset O(T \times M)$. With the new scheme, the space allocation is done in $T \times O(1) \subset O(T)$, asymptotically improving the performance of the library.

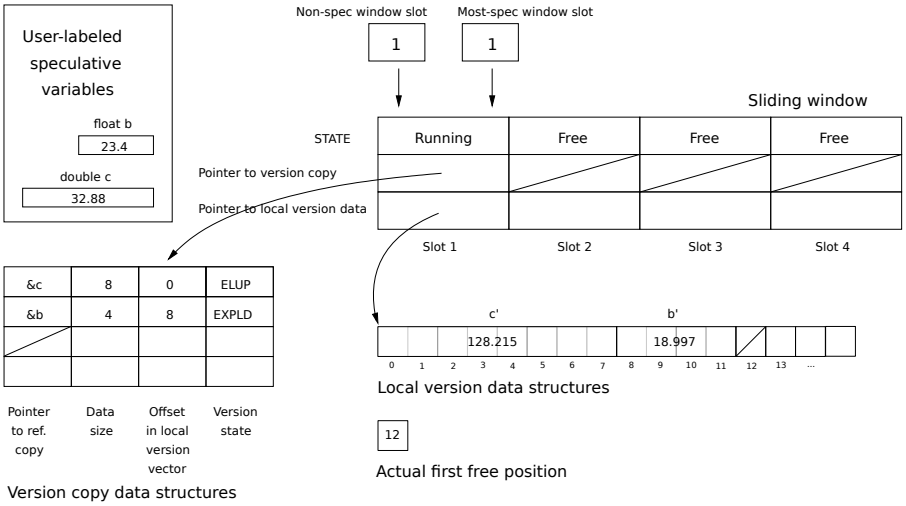

Fig. 5 Reducing operating system calls: Example with the new data structures

```
#ifdef __LP64__
    typedef unsigned long long int baseType;
#else
    typedef unsigned long int baseType;
#endif

...
baseType matrix[HASH][4][ROWS];
...
```

(a)

```
typedef struct datacell {
    void *origPointer;
    unsigned int copyOffset;
    short unsigned int size;
    short unsigned int state;
} datacell;
...
datacell matrix[HASH][ROWS]
...
```

(b)

Fig. 6 Implementation of a static example of the data structure in (a) the baseline solution, and (b) the improved version.

7.2 Structures instead of buffers

We have also modified the implementation of version copy data structures in order to improve data alignment [1] and the space needed by each version copy data structure. The baseline representation is shown in Fig. 6(a). Although the different elements in a row have different sizes, the declaration should allocate space enough to store the biggest one, in our case the pointer to the original data. This implementation requires 8 bytes for each value, that is, a total of 32 bytes for the representation of each row. Our new representation, shown in Fig. 6(b), states variables as a struct. With this representation we achieve two goals. First, we reduce the necessary memory to a half, since the memory needed to store this new structure is 16 bytes (a void pointer needs 8 bytes, an unsigned int needs 4 bytes, and each unsigned short int needs 2 bytes). Second, this structure also exploits the memory representation of C structs because these types are usually stored with the following patterns[1]: Structures between 1

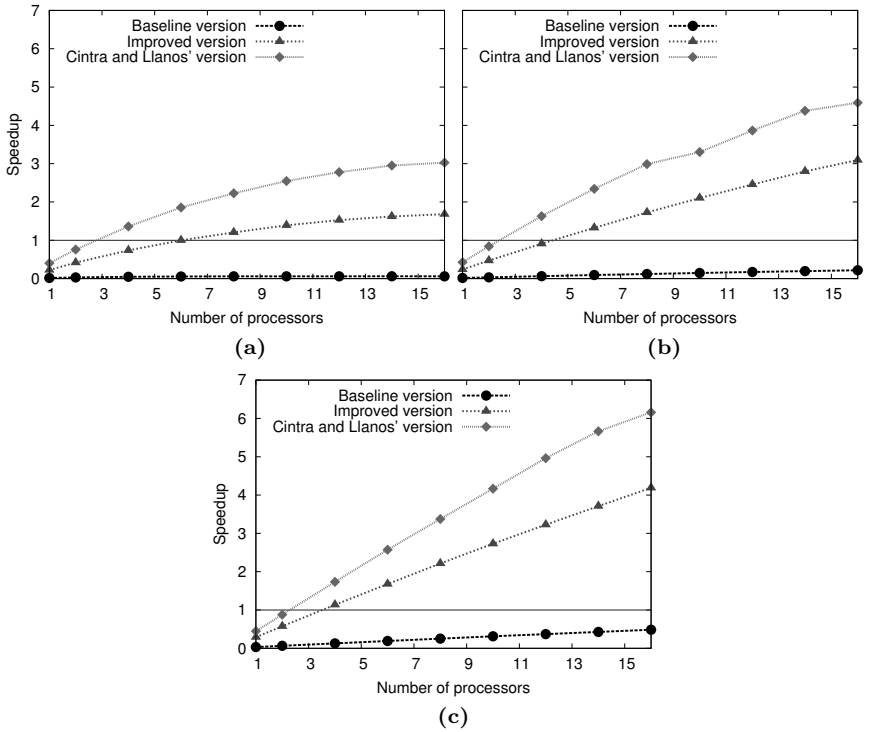

Fig. 7 Performance comparison for 2D-Hull benchmark with three different input sets: (a) Disc, (b) Square, and (c) Kuzmin.

and 4 bytes of data are usually padded so that the total structure is 4 bytes; Structures between 5 and 8 bytes of data are padded so that the total structure is 8 bytes; structures between 9 and 16 bytes of data are padded so that the total structure is 16 bytes; and structures greater than 16 bytes are padded to 16 byte boundary. A different order of the variables would add paddings because the compiler may decide to store them in 4-bytes places. Therefore, our new structure is optimized to minimize the memory space needed, thus reducing the number of cache misses.

8 Experimental results

Figs. 7 compare the performance results of the speculative parallelization of the 2D-Hull with input sets. Both the baseline version of the library and our new solution are compared. While the baseline system is not able to obtain speedups in any case, the new solution leads to a maximum speedup of 1.681× for the Disc input set (representing a 28× performance increment with respect to the baseline TLS library), 3.094× for the Square input set (14.19× performance increment) and 4.188× for Kuzmin (8.63× performance increment). To

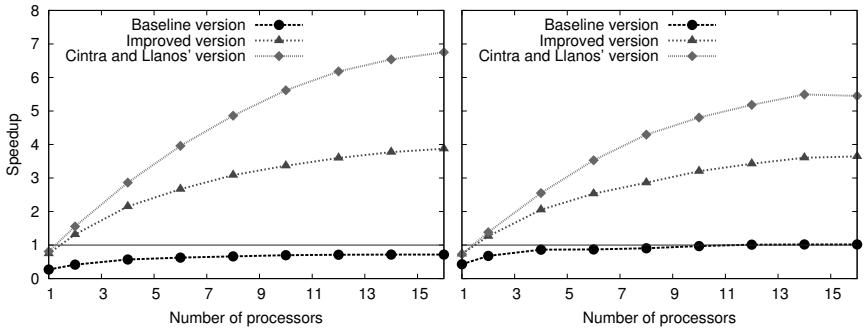

Fig. 8 Performance comparison for Delaunay benchmark with an input set of (a) 100K points, and (b) 1M points.

put these results into perspective, we also show the best speedups obtained by Cintra and Llanos with their speculative runtime system. Recall that their solution, while effective, is constrained by many limitations, and it is not generalizable to any applications. Results show that our solution allows to deliver a good percentage of the maximum speedup attainable (up to 68%), while offering a speculative solution applicable in many more cases.

Figs. 8 compares the performance results of the speculative parallelization of the Delaunay triangulation with two input sets. The new solution is again clearly better, with a maximum speedup of 3.646× for the 1M-points input set (representing a 3.58× performance increment with respect to the baseline TLS library), and 3.873× for the 100K-points input set (5.40× performance increment). Again, our solution is compared to the tailored library of Cintra and Llanos, obtaining, on average, a 75% and a 68% of their maximum speedup in the 1M-points and the 100K-points input sets, respectively.

9 Conclusions

In this paper, we have shown a solution to a problem that is common to any software-based TLS library: How to reduce search times when accessing to remote versions of speculative data. To mitigate this problem, we have implemented some optimizations such as the use of an extremely-simple hash function to avoid the need of traversing all version data, the reduction in the number of memory management system calls, and the development of new data structures to reduce memory consumption and cache misses. Our experimental evaluation with non-synthetic benchmarks on a real, shared-memory multiprocessor clearly shows that these improvements have a dramatic impact on performance: All applications tested had far better execution times than those obtained in the baseline version, and the performance results are a significant fraction of those obtained with a system specifically designed to handle these benchmarks.

Acknowledgements The authors would like to thank the anonymous reviewers for their helpful comments. The authors would also like to thank Mr. Sergio Aldea for his help in this work. This research is partly supported by the Castilla-Leon Regional Government (VA172A12-2); Ministerio de Industria, Spain (CENIT OCEANLIDER); MICINN (Spain) and the European Union FEDER (MOGECOPP project TIN2011-25639, CAPAP-H3 network TIN2010-12011-E, CAPAP-H4 network TIN2011-15734-E).

References

1. Bryant, R., David Richard, O.: Computer systems: a programmer's perspective. Prentice Hall (2003)
2. Ceze, L., Tuck, J., Torrellas, J., Cascaval, C.: Bulk disambiguation of speculative threads in multiprocessors. In: Procs of the 33rd intl symposium on Computer Architecture, ISCA '06. IEEE Computer Society, Washington, DC, USA (2006)
3. 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)
4. Cintra, M., Llanos, D.R.: Design space exploration of a software speculative parallelization scheme. IEEE Trans. on Paral. and Distr. Systems **16**(6), 562–576 (2005)
5. Cintra, M., Martínez, J.F., Torrellas, J.: Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In: Proc. of the 27th intl. symp. on Computer architecture (ISCA), pp. 256–264 (2000)
6. Clarkson, K.L., Mehlhorn, K., Seidel, R.: Four results on randomized incremental constructions. Comput. Geom. Theory Appl. **3**(4), 185–212 (1993)
7. Dai, W., An, H., Li, Q., Li, G., Deng, B., Wu, S., Li, X., Liu, Y.: A priority-aware NoC to reduce squashes in thread level speculation for chip multiprocessors. In: Procs of the 2011 IEEE 9th Int. Symposium on Parallel and Distributed Processing with Applications, ISPA '11. IEEE Computer Society, Washington, DC, USA (2011)
8. Dou, J., Cintra, M.: Compiler estimation of load imbalance overhead in speculative parallelization. In: Procs of the 13th Int. Conf. on Parallel Architectures and Compilation Techniques, PACT '04. IEEE Computer Society, Washington, DC, USA (2004)
9. Estebanez, A., Llanos, D.R., Gonzalez-Escribano, A.: Desarrollo de un motor de paralelización especulativa con soporte para aritmética de punteros. In: Proceedings of the XXIII Jornadas de Paralelismo. Elche, Alicante, Spain (2012)
10. Gao, L., Li, L., Xue, J., Yew, P.C.: SEED: A statically-greedy and dynamically-adaptive approach for speculative loop execution. IEEE Transactions on Computers **62**(5) (2013)
11. Hammond, L., Hubbert, B.A., Siu, M., Prabhu, M.K., Chen, M., Olukotun, K.: The stanford Hydra CMP. IEEE Micro **20**(2), 71–84 (2000)
12. Harris, T., Plesko, M., Shinnar, A., Tarditi, D.: Optimizing memory transactions. In: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '06, pp. 14–25. ACM, New York, NY, USA (2006)
13. Jimborean, A., Clauss, P., Dollinger, J.F., Loechner, V., Martinez Caamao, J.: Dynamic and speculative polyhedral parallelization using compiler-generated skeletons. International Journal of Parallel Programming pp. 1–17 (2013)
14. Kelsey, K., Bai, T., Ding, C., Zhang, C.: Fast track: A software system for speculative program optimization. In: Procs of the 7th annual IEEE/ACM Intl symp on Code generation and optimization, CGO '09 (2009)
15. Krishnan, V., Torrellas, J.: A chip-multiprocessor architecture with speculative multithreading. Computers, IEEE Transactions on **48**(9), 866–880 (1999)
16. Kulkarni, M., Burtscher, M., Inkulu, R., Pingali, K., Casçaval, C.: How much parallelism is there in irregular applications? In: Procs of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '09. New York, USA (2009)
17. 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: Procs of the 20th Annual Symposium on Parallelism in Algorithms and Architectures, SPAA '08, pp. 217–228. ACM, New York, NY, USA (2008)

18. Kulkarni, M., Nguyen, D., Prountzos, D., Sui, X., Pingali, K.: Exploiting the commutativity lattice. In: Procs of the 32nd ACM SIGPLAN Conf on Programming Language Design and Implementation, PLDI '11. ACM, New York, NY, USA (2011)
19. Kulkarni, M., Pingali, K., Ramanarayanan, G., Walter, B., Bala, K., Chew, L.P.: Optimistic parallelism benefits from data partitioning. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII, pp. 233–243. ACM, New York, NY, USA (2008)
20. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L.P.: Optimistic parallelism requires abstractions. In: PLDI 2007 Proceedings. ACM (2007)
21. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L.P.: Optimistic parallelism requires abstractions. Commun. ACM **52**(9), 89–97 (2009)
22. Lee, D., Schachter, B.: Two algorithms for constructing a delaunay triangulation. International Journal of Computer & Information Sciences **9**(3), 219–242 (1980)
23. M. Gupta and R. Nim: Techniques for speculative run-time parallelization of loops. Supercomputing (1998)
24. Marcuello, P., Gonzalez, A., Tubella, J.: Speculative multithreaded processors. In: Procs of the 12th Intl conference on Supercomputing, ICS '98. ACM, New York, USA (1998)
25. Mehrara, M., Hao, J., Hsu, P.C., Mahlke, S.: Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In: Procs of the 2009 conf on Prog. language design and implementation, PLDI '09. NY, USA (2009)
26. Méndez-Lojo, M., Nguyen, D., Prountzos, D., Sui, X., Hassaan, M.A., Kulkarni, M., Burtscher, M., Pingali, K.: Structure-driven optimizations for amorphous data-parallel programs. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '10, pp. 3–14. ACM, New York, USA (2010)
27. 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, USA (2009)
28. Prabhu, M.K., Olukotun, K.: Using thread-level speculation to simplify manual parallelization. In: Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '03. ACM, New York, NY, USA (2003)
29. Raman, E., Vahharajani, N., Rangan, R., August, D.I.: Spice: speculative parallel iteration chunk execution. In: Procs of the 6th annual IEEE/ACM Intl symposium on Code generation and optimization, CGO '08. ACM, New York, USA (2008)
30. Rauchwerger, L., Padua, D.: The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. SIGPLAN Not. **30**(6) (1995)
31. Sankaralingam, K., Nagarajan, R., Liu, H., Kim, C., Huh, J., Burger, D., Keckler, S., Moore, C.: Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In: Procs of the 30th Annual Intl Symp on Computer Architecture, ISCA '03 (2003)
32. Steffan, J.G., Colohan, C.B., Zhai, A., Mowry, T.C.: A scalable approach to thread-level speculation. In: Proceedings of the 27th annual international symposium on Computer architecture, ISCA '00, pp. 1–12. ACM, New York, NY, USA (2000)
33. Tian, C., Feng, M., Gupta, R.: Supporting speculative parallelization in the presence of dynamic data structures. In: Procs of the 2010 ACM SIGPLAN conf on Programming language design and implementation, PLDI '10. ACM, New York, NY, USA (2010)
34. Tian, C., Feng, M., Nagarajan, V., Gupta, R.: Copy or discard execution model for speculative parallelization on multicores. In: Procs of the 41st annual IEEE/ACM Intl Symp on Microarchitecture, MICRO '41. Washington, DC, USA (2008)
35. Tian, C., Feng, M., Nagarajan, V., Gupta, R.: Speculative parallelization of sequential loops on multicores. Int. J. Parallel Program. **37**(5), 508–535 (2009)
36. Yiapanis, P., Rosas-Ham, D., Brown, G., Luján, M.: Optimizing software runtime systems for speculative parallelization. ACM Trans. Archit. Code Optim. **9**(4) (2013)
37. Zhao, Z., Wu, B., Shen, X.: Speculative parallelization needs rigor: probabilistic analysis for optimal speculation of finite-state machine applications. In: Procs 21st Intl Conf on Parallel architectures and compilation techniques, PACT '12. New York, USA (2012)