Improving the performance of a thread-level speculation library

Alvaro Estebanez
Departamento de Informatica
Universidad de Valladolid
Valladolid, Spain 47011
Email: palestebanez1@gmail.com

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

Arturo Gonzalez-Escribano
Departamento de Informatica
Universidad de Valladolid
Valladolid, Spain 47011
Email: arturo@infor.uva.es

***Abstract*—Speculative parallelization is a technique that tries to extract parallelism of loops that can not be parallelized at compile time. The underlying idea is to optimistically execute the code in parallel, while a subsystem checks that sequential semantics have not been violated. There exist many proposals in this field, however, to the best of our knowledge, there are not any solution that allows to effectively parallelize those applications that use pointer arithmetic. In a previous work, the authors of this paper presented a software library that allow the parallelization of this kind of applications. Nevertheless, the software developed had an important limitation: Execution time of the parallelized versions was higher than the sequential one. In this work, this limitation has been addressed, finding and solving the reasons of this lack of efficiency. Experimental results obtained allow us to affirm that these limitations have been overcome.**

***Keywords*—Thread-level speculation, speculative parallelism.**

I. Introduction

DESPITE of the high speeds of sequential processors, there is the need of increase it even more. However, due to technology advances, we are beginning to reach barriers that can not be overcome. Chips are becoming more and more sophisticated. Moreover, they are full of transistors and heat dissipation is becoming a big problem. In this context, computers with more than one processor have emerged to minimize heat centralization and to perform several task at the same time, in other words, to parallelize the work. In order to exploit this kind of processors in a single code, we need programs whose instructions can be executed at the same time, i.e., programs whose instructions do not have to be executed sequentially. To decide if a sequential code fulfill this requirement is a tedious task because we need to take into account many factors to avoid synchronization errors. There exist some specific languages to develop this tasks, but some knowledge about underlying hardware and about the problem to be parallelized is still needed. In addition, it is not useful to develop software to specific architectures if it will not be portable to other machines. So, the best way to parallelize a source code is allowing compilers to do this task. At the present time, there exist some compilers that may parallelize a code. However compilers are intrinsically conservative and the result is far from optimum. The main reason of this fact is the existence of dependence violations among accesses to program data. An algorithm should have independent instructions to allow its concurrent execution, nevertheless, predict this task is not a easy. So, in fact, if compilers have the slightest suspicion of the existence of a dependence violation, they do not create parallel codes. Figure 1(a) shows a loop without dependence violations: All the instructions are independent, so, compiler may directly order its parallel execution. On the other hand, Figure 1(b) contains a loop that may produce some dependence violations.

```
for (i=0; i<MAX; i++) {
  v[i] = func(v[i]);
}
```

(a)

```
for (i=1; i<MAX; i++) {
  if (i==k)
    v[i] = func(v[i-2]);
  else
    v[i] = func(v[i]);
}
```

(b)

Fig. 1. (a) Example of a loop without dependences within its iterations. (b) Loop with dependences.

Thread-level speculation (TLS) [1], [2], [3], [4], [5], [6], [7] aims to extract loop- and task-level parallelism when a compile-time dependence analysis can not guarantee that a given sequential code is safely parallelizable. Speculative parallelization procedure assumes that sequential code can be optimistically executed in parallel, and relies on a runtime monitor to ensure that no dependence violations are produced. A dependence violation appears when a datum is produced that has been already used by a successor of the thread. In this case, results calculated so far by the successor (called the offending thread) are not valid and should be discarded, and then, this thread is restarted with the correct values. So, the less dependence violations, the better results are obtained.

There are many approaches that support TLS, however, to the best of our knowledge none of them support the use of complex structures. Therefore, in a previous work [8], [9] we developed a library that implements TLS ideas in software, however experimental results were not good enough to improve the time spent by the sequential version of the benchmarks used. So, in this paper we describe the mechanism followed to improve its performance in order to achieve better results than the sequential versions of the benchmarks.

We have improved a TLS runtime library that handles the parallel execution of loops with speculative variables, including support for speculative access of pointer-based data of any size without the need of a compile-time analysis. This runtime library not only manages accesses to speculative data, but also handles the scheduling of iterations among threads

and ensures correctness in the parallel execution of the loop. Our previous version had a bad performance, not being useful in practice. After the improvements described in this paper, experimental results lead us to affirm that new scheme is good enough to extract parallelism of both benchmarks and real programs.

The rest of this paper is structured as follows: Section II provides a perspective of the state of the art in the field of TLS. Section III introduces the initial library in order to obtain an idea of the main bottlenecks that it had. Section IV explains three different improvements applied to the library. After that, in section V we will show the experimental results achieved after the application of that optimizations. Finally, section VI draws some conclusions extracted from the analysis carried out in this paper.

II. Related work

Thread-level speculation (TLS) is an aggressive parallelization technique that is applied to regions of code that, although contain a good amount of parallelism, can not be proven at compile time to preserve the sequential semantics under parallel execution. This section reviews several thread-level speculation solutions.

The main precursor of TLS is located in the work of Rauchwerger and Padua [6]. They proposed the use of LRPD tests in conjunction with speculative parallelization of loops, in this way, loops were executed in parallel and if some errors were found, the involved loop was re-executed. This approach was based on software. Several proposals used hardware to develop a scheme to exploit TLS [10], [11], [12], [13], [14], [15]. However we are going to center our revision in software approaches because our solution is fully software-based.

In this way, Cintra and Llanos in [1] contribute with a scheme based in an aggressive sliding window, with checks for data dependence violations on speculative stores with reduced synchronization constraints.

Kelsey et *al.* developed a system called *FastTrack* that performs an unsafe optimization of sequential code [16], i.e., they created a software system that manages speculative parallelization. Specifically, their programming interface enables programming by suggestions, so, user can suggest faster implementations based on partial knowledge about a program and its usage. They divide code in two branches, the fast track and the normal track, and programmers can change between both tracks when they want. Oancea et *al.* [17] proposed a TLS system whose main design principle was to decrease overheads of speculative operations.

In 2008, Tian et *al.* in [18] proposed the Copy-or-Discard (CorD) execution model, in which the state of speculative parallel threads is maintained separately from the non-speculative computation state. If speculation is successful, the results of the speculative computation are committed by copying them into the non-speculative state. If misspeculation is detected, no costly state recovery mechanisms are needed as the speculative state can be simply discarded. Some years later, Tian et *al.* developed mechanisms that enable CorD to efficiently supports speculative execution of programs that operate on heap based linked dynamic data structures. Their are described in [19]. In particular, they proposed a copy-on-write scheme which limits the copying to only those nodes in the dynamic data structure that are modified by the speculative computation. When a speculative thread writes to a node in a dynamic data structure for the first time, the node is copied into speculative state. If a speculative thread only reads a node in the non-speculative state, it is allowed to directly access the node from the non-speculative state.

Another work of Tian et *al.* is [20]. In this paper they developed an approach for incremental recovery in which, instead of discarding all of the results and re-executing the speculative computation in its entirety, the computation is restarted from the earliest point at which a mis-speculation causing value is read. With those advances the cost of recovery is reduced as only part of the computation is re-executed, and, since recovery takes less time, the likelihood of future mis-speculations is reduced. The main idea to decouple the space allocation from thread creation is to create a new subspace when a speculate value is first read.

There exist some out-of-order engines that try to extract parallelism of sequential programs with the use of a look-ahead guide that examines "future instructions" of the program to perform a parallel execution. However, several times the guide itself is the main bottleneck of the program. In [21] Garg et *al.* use TLS to avoid some of the mentioned overheads.

Feng et *al.* developed a system that deals with efficient parallelization of hybrid loops in [22], that is, those loops that contain a mix of computation and I/O operations. They tried to get that purpose by applying DOALL parallelism to hybrid loops by breaking the cross-iteration dependences caused by I/O operations. Authors developed a support to enable speculative parallelization of hybrid loops by performing some modifications to the code. To effectively parallelize hybrid loops they developed techniques for reducing bus contention, specifically, they proposed the use of helper threading.

Fan et *al.* in [23] developed a software-based speculative framework that implements value prediction, value checking and dynamic task partition and scheduling.

Yiapanis et *al.* in [24] introduced a new structure that optimizes memory overheads of classical approaches based on the idea of mapping every user-accessed addresses into an array of integers using a hash function, and applied it to a new speculative library called *MiniTLS*.

Current researches are centered in the application of TLS to different context, i.e., Jang et *al.* [25] described a way to decompress data through the use of speculative parallelism. In this way, their algorithms are improved with the use TLS. Also Martinsen et *al.* [26] used speculative mechanism in a Web context, for this task, they developed a software through Squirrelfish JavaScript environment. The use of TLS in this context let them to achieve high levels of speedups. GPUs models are also benefited with the use of TLS, in this way, Zhang et *al.* introduces in 2013 a new library based on sliding windows that supports TLS in GPUs [27].

III. Baseline runtime library for TLS

We developed a new TLS runtime library that supports the speculative execution of for loops. A preliminary work on this

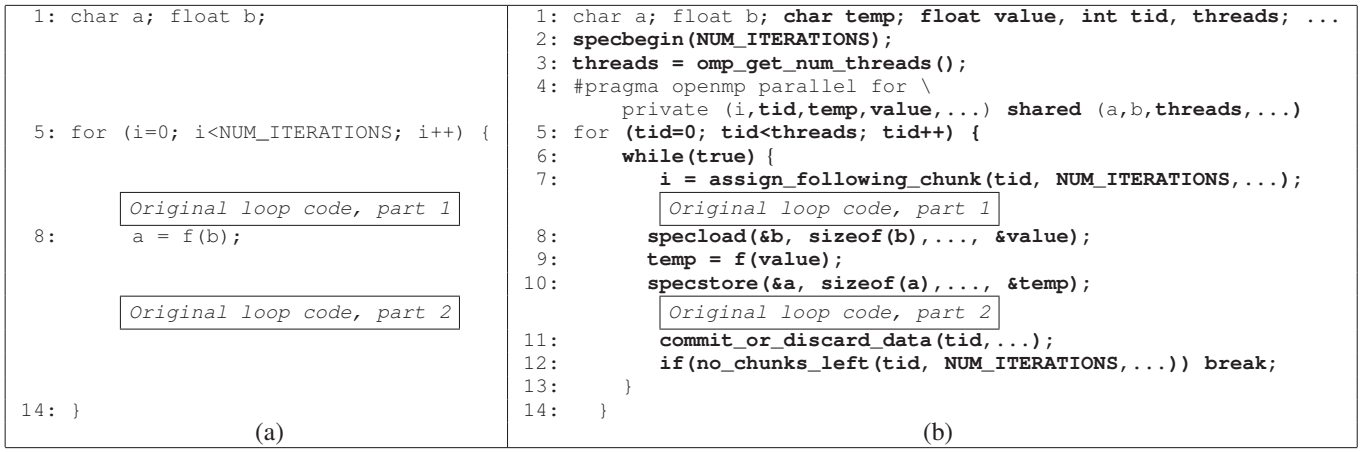

Fig. 2. Loop transformation to allow its speculative execution: Original (a) and transformed (b) code.

topic was published in [9]. The library architecture followed the design principles of the speculative parallelization library developed by Cintra and Llanos [1]. That TLS runtime library allows to speculatively access variables of any data type, both by name or by address, and managing the space needed for version copies on demand. In this section we will briefly show the general architecture of this library that will be used as the baseline to test some improvements proposed in this paper.

A. Loop transformation for speculative execution

Figure 2 briefly shows the transformation of a parallel loop that the user should carry out to allow its speculative execution. Changes are briefly described below:

Line 1: Additional, internal variables are defined.

Line 2: A call to specbegin() initializes the runtime speculative library. This speculative library uses some data structures that should be initialized before the execution of any speculative code, and the function responsible is specbegin(UINT num_iterations). This function should assign the static, or dynamic, block size that slots will use, and allocates the memory needed by dynamic data structures used by this library. It also initializes non-speculative and most-speculative pointers, states of the sliding window to the FREE state, and assigns the limit value of iterations at run-time with the use of the single integer argument used by this function.

Line 3: Before the loop, the omp_get_num_threads() function is called to obtain the number of available threads.

Line 4: Following the specifications of OpenMP [28], all variables of the loop should be classified as private or shared. In addition, internal variables needed by the runtime systems should be labeled, such as tid and threads in our example.

Line 5: The original loop structure is replaced with a parallel for loop with just "threads" iterations. This launches the number of desired threads.

Line 6: A while(true) loop ensures that each thread repeatedly requires a chunk of iterations from the original loop to be processed. If no chunks are left, a break statement exits this loop and the end of the thread is reached (see line 12).

Line 7: Inside the loop, each thread receives the index of the first iteration of its assigned chunk and proceeds with the original loop body.

Lines 8-10: The read of b variable in line 8 of Fig. 2(a) is replaced with a call to the specload() function, that recovers the most up-to-date value for this variable. The exact behavior of specload() is described later in this section. The value is stored in a private, temporal location. Line 8 of Fig. 2(a) also performs a write on a. This write is replaced with a call to specstore() (line 10 of Fig. 2(b)), that first stores the value in a local version copy and then checks whether a successor has already consumed an outdated value of a. If so, the offending thread and some or all of its successors (depending on the squash policy being defined [29]) are squashed.

It is important to highlight that only the lines of the original loop body that involve speculative variables are changed, therefore, the remaining code is left with no changes.

Line 11: Once finished the original loop body, a call to commit_or_discard_data() checks whether the thread has been squashed or not. If a squash operation was issued by a predecessor, local copies of speculative data will be discarded. If the thread has not been squashed and it is the not-spec one, a partial commit will occur. Partial commits will be described in Sect. III-D.

Line 12: After finishing their tasks related to the current chunk, all threads check whether there are no pending chunks to be executed. If there is no pending work, threads leave the while loop.

When all threads have exited the while(true) loop, the end of the parallel section has been reached and (despite the number of needed attempts) all chunks of iterations have been successfully executed, and their results committed to the speculative variables.

B. Data structures

The data structures needed by the baseline speculative library are depicted in Fig. 3(a). The sliding window mechanism is implemented by a matrix with W window slots (four in the figure). Each slot acts as a "blackboard" used to handle

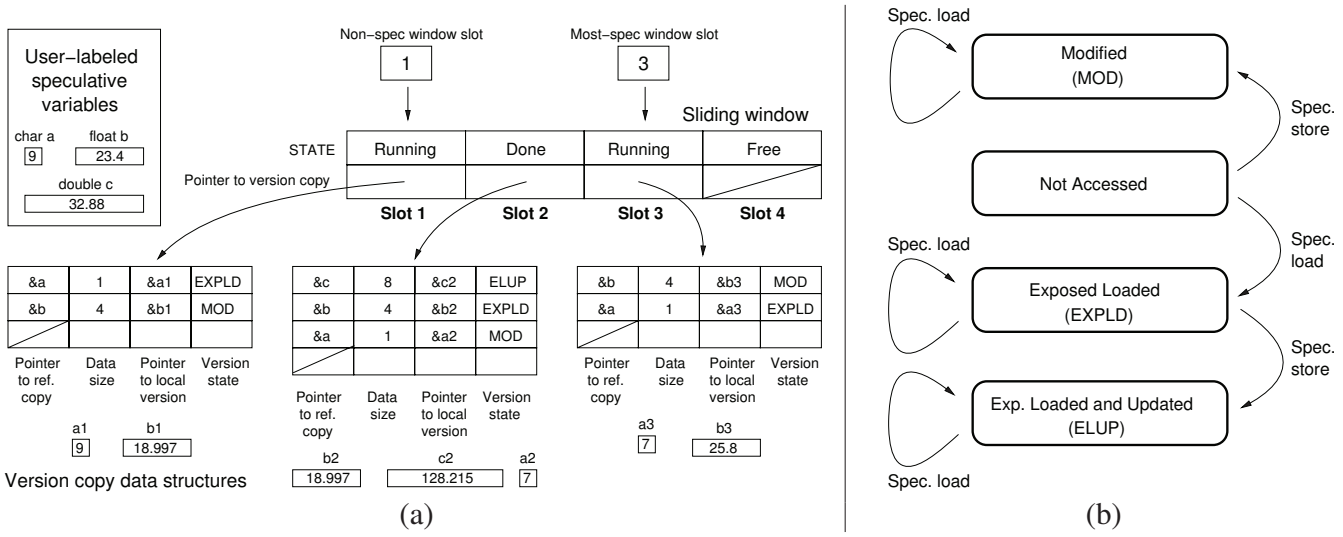

Fig. 3. (a) Data structures of our new speculative library. (b) State transition diagram for speculative data.

the speculative execution of a particular chunk of iterations. Two global variables, non-spec and most-spec, indicate the slot assigned to the execution of the non-speculative and most-speculative chunks of iterations at the moment. The STATE field indicates the state of the execution being carried out in each slot.

The figure represents the parallel execution of a loop. The loop has been divided into three chunks of iterations, and it will 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 our 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. III-D, 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. Figure 3(a) 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. The transition diagram for these states is shown in Fig. 3(b).

Figure 3(a) 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 thread working in slot 3 has forwarded its value. With respect to variable b, the information in the figure shows that b have been overwritten both by threads working in slots 1 and 3.

C. Speculative loads and stores

The interface of our implementation of specload() (see line 8 of figure 2(b)) 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.

Recall that specload() should return the most up-to-date value available for the speculative variable, to do so, the thread **traverses sequentially its data structure** and those of their predecessor to find the most up-to-date value.

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. To do so, this operation also *traverses sequentially its version copy data structure* to find and update the value, and then *searches sequentially in all successors' data structures* to find a possible RAW read-after-write) dependency. As we will see, this was the main source of inefficiencies of this library version.

D. Partial commit operation

The partial commit operation is exclusively carried out by the non-speculative thread. Every time a thread executes commit_or_discard() (line 11 of figure 2(b)), it first checks if it has not been squashed and if is the non-speculative. If the thread is speculative, the slot is left to be committed by the non-spec thread.

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, where a thread can overwrite another thread's state.

IV. Performance improvements

A. Reducing operating system calls

One of the problems detected was the excessive number of calls to the *malloc()* and *free()* functions. To better understanding the reasons, we will use an example. Suppose that a thread executes one of the main speculative functions, i.e., *specload()* or *specstore()*. In this context, thread searches in its matrix for the address of the datum being accessed. Imagine that the datum has not been used yet, so it should be added to the matrix. In this process of attaching the new data to the matrix of the thread, we have to allocate some memory to store the local copy of this datum, therefore, the *malloc()* function should be called. Specifically, each thread reserves memory with *malloc* always that works with a datum that has never been used before.

On the other hand, there is also the need of freeing all the reserved memory, therefore, threads call *free()* to free the memory used by them when they want to reuse the slot of the sliding window. Data are not freed when a slot reaches the FREE state. Instead, data are freed when a new thread is assigned to a new slot, becoming a RUN slot. Therefore, to perform this operation, all the values should be freed one by one.

Obviously, these operations spend much time because they are called very frequently. We devised that all of these operations could be changed by the implementation of a container for all the data used by each thread. Hence, a new dynamic vector was developed that allows to avoid almost all *malloc()* and *free()* operations: ***Local Version Data***.

This new vector is needed by all the available threads. We have to perform an initial call to the *malloc()* function to allocate the memory of the vector of each thread, and a final *free()* call is used to free the memory allocated. In this way, we only have to call *malloc()* again if the vector is full, and this occurs rarely. This solution greatly improves the performance observed.

However, the new structure modifies 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 approximation manages an offset for each datum. In this way, each datum will be stored in the Local Version Data vector in the position pointed by its offset, from this position to the same position plus the size of the datum, i.e., each position of the vector would store a byte, so, a datum that require four bytes to be stored would need four positions in the vector.

Also, each slot of the sliding window requires an additional pointer to the actual offset of its vector. Hence, the sliding window is augmented with another element to indicate the first free position of this new vector.

This ideas will be better understood with the use of a graphical example. Instead of having many data elements lying around (see data a1, b1, b2, etc. in Fig. 3(a), bottom) Figure 4 shows solution. In this way, suppose that a single thread is in execution (to avoid an unnecessarily complex situation) and it has managed three data, c, b and a, with a size of 8, 4 and 1 respectively. So, taking into account that this library is implemented in C language (vectors begin at 0 position), the current offset of this thread (that points to free space) will be 13.

Regarding *free()* operations, threads of the baseline solution needed to access sequentially to all the elements of its version copy, to free separately each datum, and finally to mark the first position as free. However, with the new version of the library, it is only needed to set to 0 the first free position in the local data vector, thus making the first position of the version copy as free.

Therefore, this optimization avoids the need of separate accesses to the elements of the version copy of the threads. Specifically, being T the number of threads, and N the number of data elements stored locally, this operation was initially in $T \times O(N) = O(TN)$. With the new scheme the free is done in $T \times O(1) = O(T)$, this optimization will asymptotically improve the library performance.

B. Commit optimization

Commit operations in the original library required to check all the local elements accessed by the thread, both modified, and unmodified items. So, in addition to the other modifications, we performed an optimization inspired in the work described in [1], [2], an ***Indirection Matrix***. This matrix aims to optimize commit operations because it will be used to store a list of updated elements, in order to only commit them.

With this solution, each thread will have its own vector of modified items, where positions of this vector will point to the position of the updated item. Moreover, a *tail* pointer that will point out the position of the last modified item of the *Indirection Matrix* is implemented. This new variable will ease the addition of new elements to the vector. Now, to perform the *commit* operation it is only needed to copy the elements of this list.

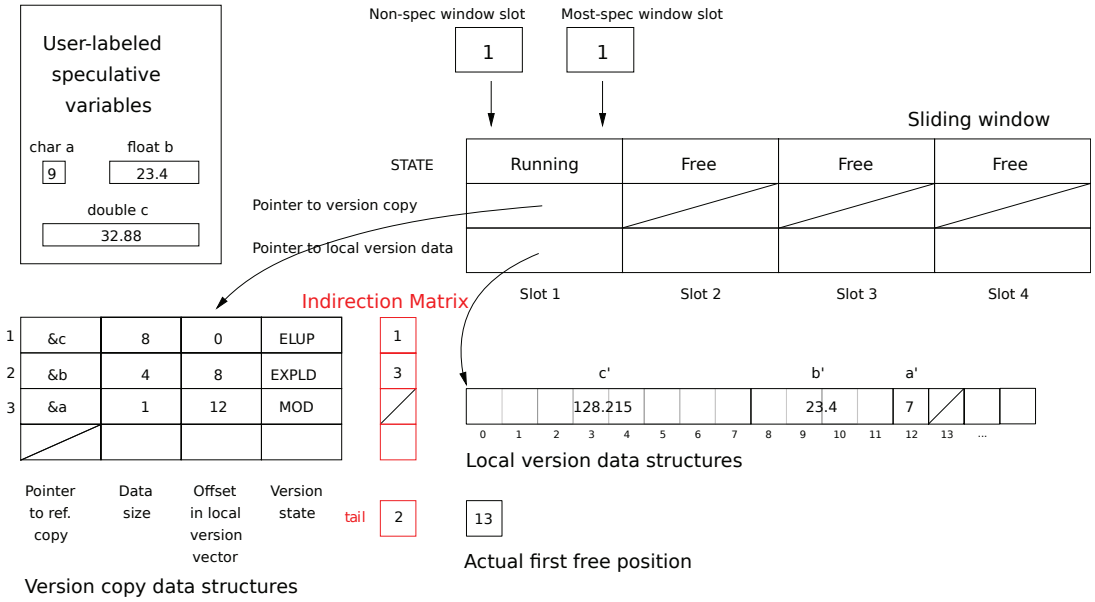

Fig. 4. Reducing operating system calls: Example with the new data structures

Figure 4 shows this new structures in the updated architecture, including the optimization of the vector with the local data. Imagine that, in the situation depicted in the example, thread finishes the execution of its chunk of iterations, as long as the thread is the non-spec, it will begin to commit its elements. Therefore, it scans its Indirection Matrix to locate those variables in ELUP or MOD state. In our example, c has been loaded and updated, and a has been modified so it would copy the content of a' into a, and the content of c' into c. In this way, the attempt of committing the content of b is avoided, unlike what happened in the original solution.

C. Reducing search time

One of the main advantages of our 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 N the number of data elements stored locally, the search is done in $T \times O(N) = O(TN)$. Therefore, the performance figures for our library with this mechanism were severely limited.

The main bottleneck comes from the sequential checks performed during *specload()* and *specstore()* functions, where each element should be inspected to be compared with one of the arguments, to detect if it has been used before, or to get the most updated value of it. 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 have developed an alternative structure with X tables.

Before accessing the data, an AND operation on the address of the user-defined speculative variable obtains a hash H, in the range $0 \ldots (X-1)$. This hash is used to look into the Hth tables 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 $O(\frac{T.N}{H})$ search times. Note that, while N is a relatively small number (typically up to 64 for current shared memory architectures), it can be set as big as needed.

Figure 5(right) shows the new 3D version copy structure with an example. Suppose that the baseline version copy of a thread has the values depicted on the left of the figure. With the new 3D structure, this version copy is transformed into the structure shown on the right of the picture, supposing that the hash of c and b is 0, and the hash of a is X-1. Also, the first value of each hash row that has not been previously used, is marked as void. The same occurs with the third position (depth) of the 0 hash position, and the second position (depth) of the X-1 hash position.

In the practice, these ideas have been implemented with the mentioned third dimensional structure but, in order to have a better understanding of this optimization, another point of view could be used: Imagine the structure with three dimensions as if exists a vector with H positions where each one of them have H pointers to H version copy structures, i.e., instead of using a version copy for each slot, use H version copies for each slot. This idea, conceptually similar to the one explained in the previous paragraph and the same example that is shown in Figure 5, is depicted in Figure 6.

The concepts introduced by the first optimization continue to apply, so, a single local version data structure is used by each slot. In this way, variables used in Figures 5 and 6 preserve the same offset values in the version without the third optimization and in the new version.

On the other hand, the second optimization, that is, include an Indirection Matrix to the schema, can not be used without the

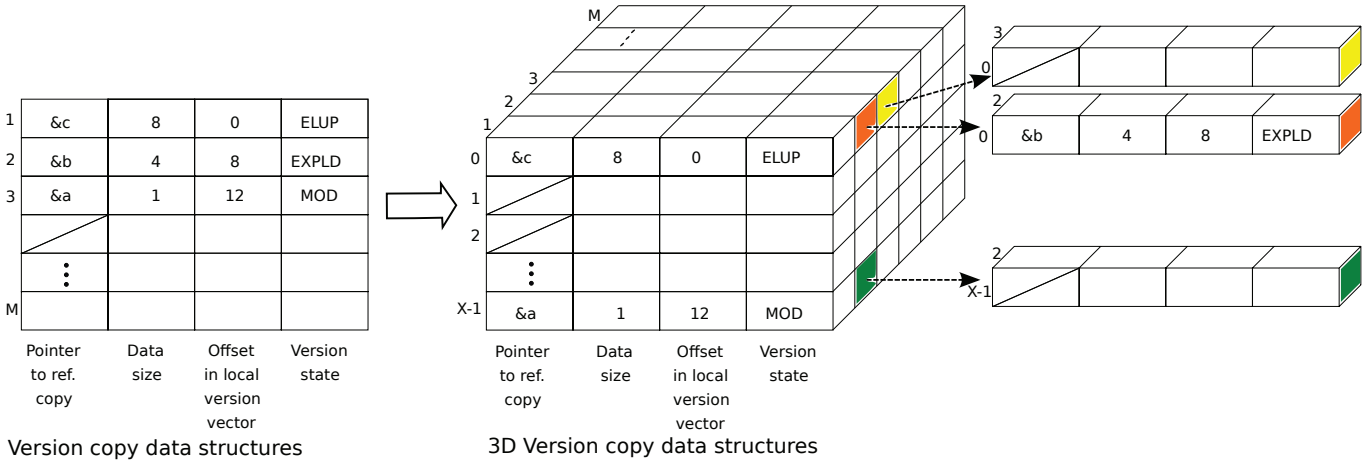

Fig. 5. Optimization 3: Structures with three dimensions.

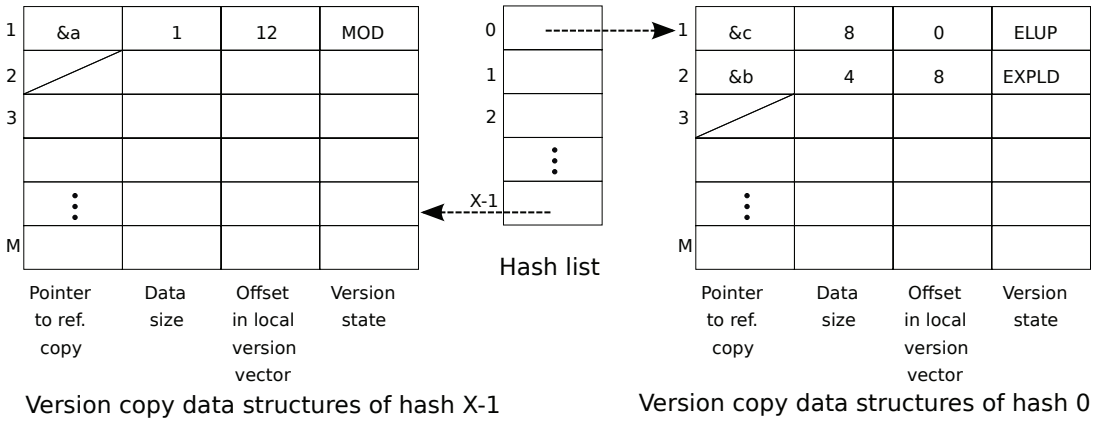

Fig. 6. Optimization 3: Structures with three dimensions, another point of view.

application of some modifications to allow that the mentioned matrix has the same functionality than the previous one.

D. Implementation issues

As we stated previously the new version of the speculative parallelization library uses an additional dimension to improve performance, therefore, the *Indirection Matrix* also needs to include an additional dimension to its structure. In the previous version of the scheme a single version copy was managed by each slot, consequently, if no changes were performed in the mentioned matrix, each position of the *Indirection Matrix* will point to a single version copy instead of taking into account existing H version copies. On the other hand, perform changes in the basis of the *indirection matrix* by including the hash position H in order to maintain a single vector, instead of the position of the datum in the version copy, is not desirable because a hash position will point to several data and some of them will not be modified.

Another problem detected when we followed this approach is that the same row will be attached several times in the same column of the mentioned matrix. This case will be better understood with the use of an example. Suppose that a thread update a datum from the *hash* position 30, that points to the address 5 000 of the memory. In order to follow the semantics of the *specstore()* operation, this thread will add the datum to its matrix in the 30*th hash* position. Finally, this *hash* position is added to the *Indirection Matrix*, and the pointer to the last data of this matrix is augmented. On the other hand, suppose that in the next operation, the same thread update another datum, with the same *hash*, 30, but now, the address pointed is, for example, 6 000. Then the datum will be added to the matrix of the thread, and then, erroneously the *hash* position of this datum, 30, is attached again to the *Indirection Matrix*.

Hence, we should attach a new dimension to the mentioned *Indirection Matrix*. This implementation will allow to commit only the elements of the *hash* position that have been used, instead of all of them. In this way, each one of the version copies used will have its own *indirection matrix*, and the process of adding a datum to this matrix is similar than the previous method, with the only difference that now, H hash positions are used to obtain the third dimension of this new *indirection cube*.

By extension, it is needed to add a new dimension to the tail position of the indirection matrix because each H hash position of the structure has its own number of modified items.

V. Experimental results

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.

To test the new library three synthetic benchmarks and the 2-dimensional Convex Hull problem (2D-Hull) have been used. We will compare four different versions:

- **Original version (*v-40*):** This is considered the initial version of the library, that is, the version without any optimizations.
- **Indirection Matrix Version (*v-43*):** This version implements one of the described optimizations, the Indirection Matrix.
- **Two-dimensional version with no system calls (*v-44*):** This version incorporates the Indirection Matrix, and a second optimization, it removes all *malloc()* and *free()* functions related to memory allocation of local variables of the threads.
- **Three-dimensional version with no system calls (*v-45*):** The last version used implements Indirection Matrices, removes all the functions mentioned in the previous point, and implements a three-dimensional structure in order to avoid the sequential access to all the local elements of the threads.

A. Convex hull

In this case, two different set of points have been used, both composed of 10 000 000 points, one that contains a uniformly-distributed, square-shaped set of points, and another that contains a uniformly-distributed, disc-shaped set of points that follows a Kuzmin distribution. Sequential accesses was a big bottleneck because this application used a big number of speculative variables. In this way, other versions had to perform accesses element by element, and their results are worse than the three-dimensional version (*v-45*).

Figure 7 shows the results obtained with the square-shaped input set. Execution times achieved with this version are on average a 421.4% faster than those obtained with the original version.

On the other hand, Figure 8 summarizes the results obtained with the disc-shaped input set. Execution times achieved with this version are on average a 314.7% faster than those obtained with the original version.

B. Synthetic benchmarks evaluation

Figure 9 shows the code of three synthetic benchmarks: Complete, Tough and Fast.

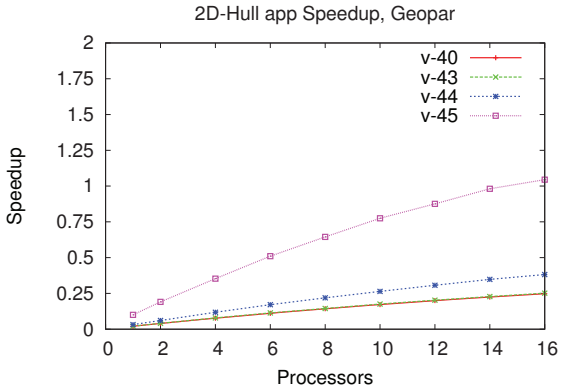

Fig. 7. Speed-up obtained after executing 2D-Hull with the speculative library with the Square input set.

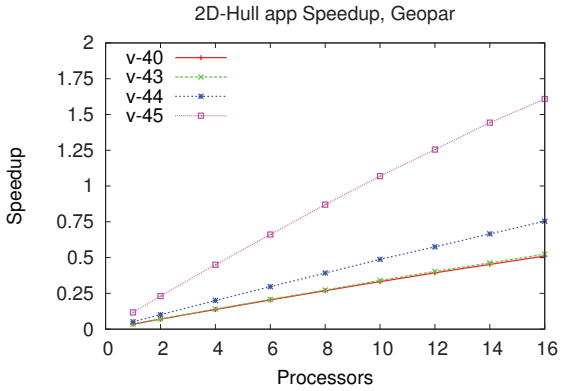

Fig. 8. Speed-up obtained after executing 2D-Hull with the speculative library with the Kuzmin input set.

The Complete benchmark, shown in Fig. 9(a), aims to concurrently test the most useful features of our solution, including (1) speculative access of data with different sizes, and (2) speculative access to data structures. While executing this loop in parallel, all the iterations lead to dependence violation. Consequently, the speedup obtained is extremely poor, but the parallel execution finishes successfully. Figure 10(a) depict the speed-up obtained with this application.

The Tough benchmark, depicted in Fig. 9(b), was designed to heavily test the robustness of our solution and of the underlying consistency protocol used. All of its iterations perform a load and a store on the same speculative data structure, with almost no computational load on private variables. This situation adversely affects performance, although the number of dependence violations during parallel execution is relatively small (4.46%). Despite of this, the parallel execution is also successful. Figure 10(b) depicts the speed-up obtained with this application.

Finally, the Fast benchmark, shown in Fig. 9(c), has been designed to test the efficiency of the speculative scheduling mechanism. In this benchmark, only one of the 30 000 iterations (0.003%) lead to a dependence violation. Note that this single dependence is enough to prevent the compile-time parallelization of this loop. The parallel execution of this

```
C00: #define NITER 6000
C01: int array[MAX], array2[MAX];
C02: struct card{ int field; };
C03: struct card p1 = {3}, p2 = {99999}, p3 = {11111};
C04: char aux_char = 'a';
C05: double aux_double = 3.435;
C06: int i, j;
       ...
C07: #pragma omp parallel for default(none) \
C08:   private(i,j) shared(array1,p2) \
C09:   speculative(p1,p3,aux_char,aux_double,array2)
C10: for ( i = 0 ; i < NITER ; i++ ) {
C11:   for ( j = 0 ; j < NITER ; j++ ) {
C12:     if (i <= 1000) p1.field = array[i%4] + j;
C13:     else array2[i%4] = p1.field;
C14:     if (i > 2000) aux_char = i%20 + 48 + aux_char%48;
C15:     else aux_char = i%20 + array[i%4]%10 + 48;
C16:     if (i > 1500)
C17:       aux_double = array[i%4]/(i+1) + aux_double;
C18:     else array2[i%4] = (int) (aux_double / i*j) + \
           (array2[(i+j)%4] +i*j)%1234545;
C19:     if (i*j > 10000) p1 = p2; else p3 = p1;
C20:   }
C21: }
```

(a)

```
T00: #define NITER 1000000, MAX 100
T01: int array[MAX];
       ...
T02: #pragma omp parallel default(none) \
T03:   private(P) \
T04:   speculative(array)
T05: for ( P = 0 ; P < NITER ; P++ ) {
T06:   Q = P % (MAX) + 1;
T07:   aux = array[Q-1];
T08:   Q = (4 * aux) % (MAX) + 1;
T09:   array[Q-1] = aux;
T10: }
```

(b)

```
F00: #define NITER 30000
F01: int array[MAX];
F02: int i,j,k;
F03: int spec1=0, spec2=0;
F04: int iter1, iter2;
       ...
F06: #pragma omp parallel default(none) \
F07:   private(i,k) shared(array,iter1,iter2) \
F08:   speculative(spec1,spec2)
F09: for ( i = 0 ; i {s< NITER ; i++ ) {
F10:   if (i == iter1) j = spec1;
F11:   if (i == iter2) j = spec2;
F12:   for (k = 0; k<array[i%MAX]+j; k++) {
F13:     if (k >= 29900)
F14:       spec1 = (k + array[(i+k)%MAX] ) \
             % NITER;
F15:     if (k <= 200) spec2 = array[i%MAX];
F16:   }
F17:   if (i == NITER-1) spec1 = spec2;
F18: }
```

(c)

Fig. 9. Synthetic benchmarks used: (a) Complete; (b) Tough; (c) Fast.

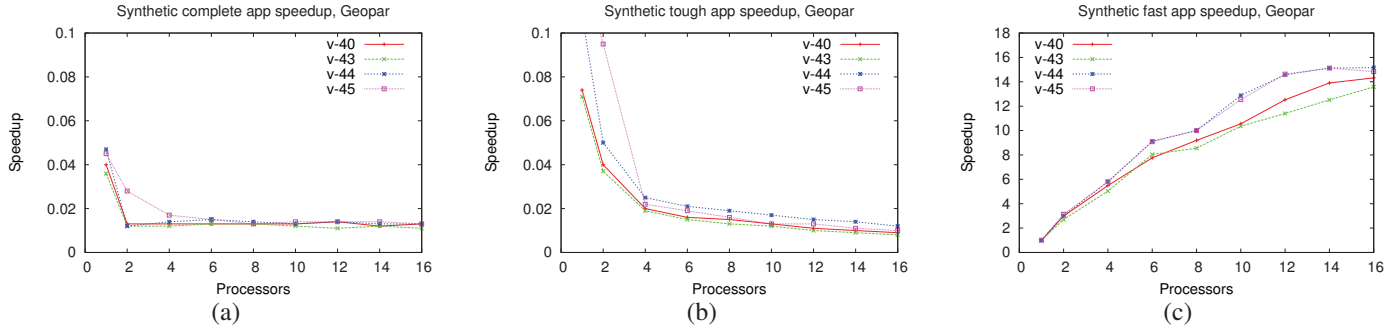

Fig. 10. (a) Speed-up obtained after executing synthetic benchmarks with the speculative library. (a) Complete, (b) Tough and (c) Fast.

benchmark with 16 processors leads to a 15.16× speedup with the version *v-44*. The obtained efficiency, 94.75% for 16 threads, indicates that the overhead due to the speculative scheduling mechanism itself is negligible. Figure 10(c) depicts the speed-up obtained with this application.

VI. Conclusions

In this paper, we have shown how we have improved the performance of our pointer-based TLS library. To do so, we have implemented some optimizations such as the reduction in the number of memory management system calls, and the replacement of data structures to avoid their sequential traversing. In this way, experimental results in terms of execution time clearly show that the improvements applied to the library have a direct impact on performance: All applications tested had better execution times than those obtained in the previous version [8], [9]. Therefore, experimental results lead us to conclude that current version of the engine is faster that the original one.

This work is result of a Master Thesis [30].

Acknowledgments

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

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

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

[4] D. R. Llanos, "Thread–level speculative parallelization," in *Science and Supercomputing in Europe, 2004 Annual Report*, P. Alberigo, G. Erbacci, and F. Garofalo, Eds. Italy: CINECA, 2005, pp. 211–213, iSBN 88-86037-15-5.

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

[6] L. Rauchwerger and D. Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization," *SIGPLAN Not.*, vol. 30, no. 6, pp. 218–232, Jun. 1995. [Online]. Available: http://doi.acm.org/10.1145/223428.207148

[7] A. Garcia-Yaguez, D. R. Llanos, and A. Gonzalez-Escribano, "Robust thread-level speculation," in *Proceedings of the 2011 18th International Conference on High Performance Computing*, ser. HIPC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/HiPC.2011.6152737

[8] A. Estebanez, D. R. Llanos, and A. Gonzalez-Escribano, "Desarrollo de un motor de paralelización especulativa con soporte para aritmética de punteros," Trabajo de Fin de Grado, Universidad de Valladolid, July 2012.

[9] ——, "Desarrollo de un motor de paralelización especulativa con soporte para aritmética de punteros," in *Proceedings of the XXIII Jornadas de Paralelismo*, Elche, Alicante, Spain, September 2012.

[10] S. Gopal, T. N. Vijaykumar, J. Smith, and G. Sohi, "Speculative versioning cache," in *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*, 1998, pp. 195–205.

[11] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A scalable approach to thread-level speculation," in *Proceedings of the 27th annual international symposium on Computer architecture*, ser. ISCA '00. New York, NY, USA: ACM, 2000, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/339647.339650

[12] M. Cintra, J. F. Martínez, and J. Torrellas, "Architectural support for scalable speculative parallelization in shared-memory multiprocessors," in *Proc. of the 27th intl. symp. on Computer architecture (ISCA)*, June 2000, pp. 256–264.

[13] K. Olukotun, L. Hammond, and M. Willey, "Improving the performance of speculatively parallel applications on the Hydra CMP," in *Proceedings of the 13th international conference on Supercomputing*, ser. ICS '99. New York, NY, USA: ACM, 1999, pp. 21–30. [Online]. Available: http://doi.acm.org/10.1145/305138.305155

[14] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas, "Tasking with out-of-order spawn in TLS chip multiprocessors: Microarchitecture and compilation," in *Proceedings of the 19th annual international conference on Supercomputing*, ser. ICS '05. New York, NY, USA: ACM, 2005, pp. 179–188. [Online]. Available: http://doi.acm.org/10.1145/1088149.1088173

[15] W. Dai, H. An, Q. Li, G. Li, B. Deng, S. Wu, X. Li, and Y. Liu, "A priority-aware NoC to reduce squashes in thread level speculation for chip multiprocessors," in *Proceedings of the 2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*, ser. ISPA '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 87–92. [Online]. Available: http://dx.doi.org/10.1109/ISPA.2011.21

[16] K. Kelsey, T. Bai, C. Ding, and C. Zhang, "Fast track: A software system for speculative program optimization," in *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*, march 2009, pp. 157 –168.

[17] C. E. Oancea, A. Mycroft, and T. Harris, "A lightweight in-place implementation for software thread-level speculation," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, ser. SPAA '09. New York, NY, USA: ACM, 2009, pp. 223–232. [Online]. Available: http://doi.acm.org/10.1145/1583991.1584050

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

[19] C. Tian, M. Feng, and R. Gupta, "Supporting speculative parallelization in the presence of dynamic data structures," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010.

[20] C. Tian, C. Lin, M. Feng, and R. Gupta, "Enhanced speculative parallelization via incremental recovery," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 189–200. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941580

[21] A. Garg, R. Parihar, and M. C. Huang, "Speculative parallelization in decoupled look-ahead," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 413–423. [Online]. Available: http://dx.doi.org/10.1109/PACT.2011.72

[22] M. Feng, R. Gupta, and I. Neamtiu, "Effective parallelization of loops in the presence of I/O operations," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 487–498. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254122

[23] X. Fan, S. Li, and W. Zhiying, "HVD-TLS: A novel framework of thread level speculation," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*, 2012, pp. 1912–1917.

[24] P. Yiapanis, D. Rosas-Ham, G. Brown, and M. Luján, "Optimizing software runtime systems for speculative parallelization," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 39:1–39:27, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400698

[25] H. Jang, C. Kim, and J. W. Lee, "Practical speculative parallelization of variable-length decompression algorithms," in *Proceedings of the 14th ACM SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, ser. LCTES '13. New York, NY, USA: ACM, 2013, pp. 55–64. [Online]. Available: http://doi.acm.org/10.1145/2465554.2465557

[26] J. Martinsen, H. Grahn, and A. Isberg, "Using speculation to enhance javascript performance in web applications," *IEEE Internet Computing*, vol. 17, no. 2, pp. 10–19, Mar. 2013. [Online]. Available: http://dx.doi.org/10.1109/MIC.2012.146

[27] C. Zhang, G. Han, and C.-L. Wang, "GPU-TLS: An efficient runtime for speculative loop parallelization on GPUs," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, 2013, pp. 120–127.

[28] "OpenMP specification, version 4.0," http://www.openmp.org/mp-documents/OpenMP_4.0_RC2.pdf.

[29] A. García-Yágüez, D. R. Llanos, and A. Gonzalez-Escribano, "Squashing alternatives for software-based speculative parallelization," *IEEE Transaction on Computers*, to appear.

[30] A. Estebanez, "Improving the Perfomance of a Pointer-Based, Speculative Parallelization Scheme," Master's thesis, University of Valladolid, Spain, July 2013.