Squashing Alternatives for Software-Based Speculative Parallelization

Álvaro García Yágüez, Diego R. Llanos, *Senior Member*, *IEEE*, and Arturo Gonzalez-Escribano

Abstract—Speculative parallelization is a runtime technique that optimistically executes sequential code in parallel, checking that no dependence violations arise. In the case of a dependence violation, all mechanisms proposed so far either switch to sequential execution, or conservatively stop and restart the offending thread and all its successors, potentially discarding work that does not depend on this particular violation. In this work we systematically explore the design space of solutions for this problem, proposing a new mechanism that reduces the number of threads that should be restarted when a data dependence violation is found. Our new solution, called exclusive squashing, keeps track of inter-thread dependencies at runtime, selectively stopping and restarting offending threads, together with all threads that have consumed data from them. We have compared this new approach with existent solutions on a real system, executing different applications with loops that are not analyzable at compile time and present as much as 10% of inter-thread dependence violations at runtime. Our experimental results show a relative performance improvement of up to 14%, together with a reduction of one-third of the numbers of squashed threads. The speculative parallelization scheme and benchmarks described in this paper are available under request.

Index Terms—Thread-level speculation, optimistic parallelization, loop-based parallelization

1 Introduction

Speculative parallelization (SP), also called Thread-Level Speculation (TLS) [1]–[3] or Optimistic Parallelization [4], [5] aims to automatically extract loop- and task-level parallelism when a compile-time dependence analysis cannot guarantee that a given sequential code is safely parallelizable. Speculative parallelization optimistically assumes that the code can be executed in parallel, and relies on a runtime monitor to ensure that no dependence violation is produced. As long as not many dependence violations arise, speculative parallelization can speed up these non-analyzable fragments of code.

Speculative parallelization can be either implemented in hardware or software. While hardware mechanisms do not need changes in the code and do not add overheads to speculative execution, they require changes in the processors and/or the cache subsystems (see e.g. [6]–[10]). Software-based speculation, on the other hand, requires to augment the original code with instructions that drive the parallel execution and the runtime dependence analysis. Although these instructions imply a performance overhead, software-based SP can be effectively used in current shared-memory systems with no hardware changes.

In a given stage of the speculative execution, different threads cooperate by executing fragments of the code in parallel. The thread executing the earlier fragment of code according to sequential semantics is called *non-speculative*, while the thread executing the last fragment is called the *most speculative*. A dependence violation appears when a given thread generates a datum that has already been consumed by a successor. In this case, the results calculated so far by this successor (called the *offending thread*) are not valid and should be discarded. It is easy to see that the risk of consuming a wrong value increases from the non-speculative to the most-speculative thread.

If the runtime monitor detects a dependence violation, a corrective operation should be carried out to let the execution progresses. Early proposals [3], [11] stop the parallel execution and restart the loop serially. A more sophisticated solution is to stop the offending thread, re-executing it with the correct value. As long as some successors of the offending thread may have consumed a value from it, usually all successors are stopped and restarted as well (see, e.g. [1], [2], [12], [13]).

This contribution goes one step further on the choice of threads that should be re-executed if a dependence violation arises. Our proposal keeps track of inter-thread dependencies, in order to only re-execute threads that are known to have consumed values from the offending thread. Being P the number of threads used, the spatial complexity of this solution is just P^2 memory words. Regarding temporal complexity, our proposal has the same complexity as conservative approaches regarding speculative loads and stores, that are in $O(P)$. If a dependence violation is found, the cost of discarding threads is in $O(P^2)$ instead of $O(P)$. We have implemented this solution upon an optimized version of a software-based speculative parallelization scheme [1], [14], and evaluate it with six different applications that present loops whose data dependences cannot be analyzable at compile time, because of the use of subscripted subscripts, complex interprocedural data flow, or input-dependent data and control flow. Our

• *The authors are with the Departamento de Informática, ETS Informática, Universidad de Valladolid, Campus Miguel Delibes, 47011 Valladolid, Spain. E-mail: alvarga87@gmail.com, {arturo, diego}@infor.uva.es.*

Manuscript received 12 Nov. 2012; revised 14 Feb. 2013; accepted 15 Feb. 2013. Date of publication 05 Mar. 2013; date of current version 27 June 2014. Recommended for acceptance by R. Gupta.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2013.46

experimental results show that keeping inter-thread dependences leads to a relative performance improvement of up to 14%, together with a reduction of 35% in the number of squashed threads comparing with the squashes produced when discarding the offending thread and all its successors.

The rest of the paper is organized as follows. Section 2 discusses the fundamentals of software-based speculative parallel execution. Section 3 analyzes in detail the design space of the squash operation, showing all the possible alternatives. Section 4 describes the implementation of the software-based, speculative parallelization scheme that will be used to evaluate the new mechanism proposed. Section 5 shows the algorithms used by the original speculative parallelization scheme to implement the squash operation. Section 6 presents the new algorithms needed to implement the new squashing proposal. Section 7 discusses the asymptotical complexity of the new mechanism proposed. Section 8 presents a detailed performance comparison of the squashing alternatives studied. Section 9 describes related work, while Section 10 concludes the paper.

2 Software-Based Speculative Parallelization

Unlike hardware-based mechanisms, software-based speculative parallelization does not need any changes in processors or caches. Instead, it allows to optimistically execute a loop in parallel using commodity shared-memory systems. To detect dependence violations, the code is augmented at compile time with functions that monitor the parallel execution, obtain the most recent value for a given variable and stop and re-start threads that have consumed wrong values. This technique makes possible to generate automatically a parallel version of a sequential loop, opening the possibility of obtaining speedups in a parallel system without the development cost of a manual parallelization, even in the presence of dependence violations.

The runtime monitor checks for dependence violations by tracking all access to the speculative data. The possible data dependencies are WAR (Write-after-Read), WAW (Write-after-Write) and RAW (Read-After-Write), and a dependence violation occurs if these dependencies execute out-of-order. WAR and WAW dependence violations can be effectively handled at runtime by using versions of data and forwarding mechanisms that will be described below. A RAW dependence violation occurs when a thread prematurely loads a datum that later is modified by a predecessor. In this case, the thread executing the latter iteration should be stopped, its partial results discarded, and re-executed using the correct values. This is known as a *squash* operation. The squash should also be extended, at least, to any successors of the consumer thread that have consumed a value from it.

To simplify squashes and avoid rollbacks, threads that execute each chunk of iterations are not allowed to change the shared data directly. Instead, each thread maintains a *version* of the shared structure. Only when the execution of the block of iterations succeeds, a *commit* operation reflects changes to the original shared structure. This operation should be done preserving the total order for each block of iterations, from the non-speculative thread to the most-speculative one. In the case of a squash operation, the data versions of all squashed threads are simply discarded. A detailed description of an example of thread-level speculation runtime management can be found at [14].

To take advantage of a speculative parallelization scheme, some simple adjustments, within the capabilities of modern compilers, have to be done to the sequential code. A summary of these changes follows.

2.1 Changes in the Loop Structure and Chunk Scheduling

Several threads should be spawn to execute the loop in parallel. Each thread receives an initial chunk of consecutive iterations. After a successful computation of this chunk, threads acquire a new one according to a given scheduling strategy. The problem of finding the optimal chunk size for parallel and speculative execution, together with the choice of an appropriate scheduling strategy, have been extensively studied in the literature. Solutions go from the *Fixed-Size Chunking* (FSC) strategy [15], where all blocks have the same size and the optimal chunk size has to be found in advance, to the *Unit Policy* where each chunk has size one (default policy in [16]). It is also possible to design a particular scheduling mechanism for a specific problem or even for a specific input set. This approach can be as challenging as designing parallel code for the problem, since it needs a deep understanding of the parallelization mechanism in order to predict and try to avoid dependence violations.

2.2 Speculative Loads

As far as each thread maintains its own version of the shared structure, all original reads to this shared data should be augmented with code that searches backwards for the most up-to-date version of the value being read. This operation is known as *forwarding*. If no predecessor owns the value, the main copy of the value is used.

2.3 Speculative Stores

Any modification to a version of the shared structure performed by a thread may lead to a dependence violation, if a successor thread has already forwarded an outdated value. Therefore, all writes to the shared structure should be augmented with code that searches forwards for threads that may have consumed a wrong value. If such threads are found, a squash operation will place and their execution will be restarted to consume the updated value.

2.4 Commit and Thread Management

After executing a block of iterations, each thread should check that it has not been squashed and performs the commit when appropriate. Instead of performing one final commit if the entire parallel execution succeeds, a common solution is to use a *sliding window* mechanism [6], [2] to hold the version data of the chunks being speculatively executed. When the non-speculative thread finishes its chunk, an in-order, partial commit takes place and the sliding window is advanced.

3 Design Space of the Squash Operation

If a dependence violation occurs, the runtime monitor should decide what to do with the parallel execution. The design space for squashing policies can be resumed in the following options.

3.1 Stops Parallel Execution

The first, most conservative solution is to simply discard the parallel work done so far, restarting the loop serially. This is the approach followed in speculative parallelization pioneering works [3], [11]. If the runtime monitor does not detect any dependence during the speculative execution, data is committed at the end of the loop. As far as this mechanism does not tolerate even a single dependence violation during parallel execution, it is only suitable for loops that cannot be analyzed at compile time, but that are expected to be fully parallelizable [17].

3.2 Inclusive Squashing

This policy stops and restarts the first offending thread found and all its successors, regardless of the data consumed by them. We call this policy *inclusive squashing*, since all threads that are more speculative than the offending thread are included in the squash. Once the offending thread is restarted, it is labeled as the new most-speculative thread, and all discarded successors should get subsequent chunks of iterations. This mechanism has been extensively implemented in both software-based (e.g. [1], [2], [12]) and hardware-based (e.g. [6], [18]) speculative parallelization schemes, allowing the parallel execution of loops even in the presence of dependence violations. The inclusive squash strategy keeps the sliding window "compact", since all threads inside the window are guaranteed to be valid. This solution simplifies the forwarding operations and the search for dependence violations at runtime, at the cost of discarding potentially valid work.

The inclusive squashing policy admits two variants. The first one, that we will call *eager inclusive squashing*, checks for dependence violations after every speculative store. The advantage of this solution is that offending threads are detected and squashed very quickly, thus reducing the total amount of work being discarded. However, this check for violations imposes an overhead to the speculative store operation. The second variant, that we will call *lazy inclusive squashing*, postpones the check for violations to commit time. Before committing, the non-speculative thread checks for threads that have incorrectly consumed values changed in the window slot being committed. This second variant leads to faster speculative store operations, at the cost of slower commits and more work discarded.

3.3 Exclusive Squashing

In this paper we propose an *exclusive squashing* mechanism, where only offending threads and all their successors that have consumed *any value* generated by them are discarded. Note that we do not need to keep the exact dependence graph for all speculative variables being loaded and stored, but just the dependence between producer and consumer threads. Being P the number of threads, this solution leads to a spatial complexity for the data structure needed of $1 \times P^2$. Note that, with this solution, the sliding window may contain "bubbles" of squashed threads whose data are not valid. This fact complicates speculative operations such as forwarding of updated values, search for violations or thread scheduling. Depending on the application, this solution leads to a significant reduction of squashed threads, at the cost of maintaining these inter-thread dependencies at runtime, regardless of the number of dependence violations effectively produced.

As in the case of the inclusive squashing, exclusive squashing also admits two variants: *eager exclusive squashing*, where the check for dependence violations is carried out after every speculative store, and *lazy exclusive squashing*, where the check is postponed until the non-speculative thread commits the values of each block of iterations successfully calculated.

3.4 Perfect Squashing

There is a fourth possibility in this design space: Squashing only offended threads and all their successors that have consumed *wrong values* generated by them. This solution implies to keep track of the dependence graph of every single speculative variable being read or written. Our experimental results suggest that this analysis is too costly to be carried out by software-only monitors, not compensating the further reduction in the number of squashes. Therefore, we believe that perfect squashing is unlikely to speed up the parallel execution in software-based TLS systems.

4 Architecture of a Software-Based TLS System

Our exploration of the squashing policies design space has been built upon an optimized implementation of the software-based speculative parallelization scheme described in [1]. This scheme performs all the tasks described in Section 2, including the scheduling of blocks of iterations, speculative loads and stores, and commits of speculative data. The original version of this speculative scheme only supports eager inclusive squashing. In this section we will describe the data structures needed to handle speculative execution in this baseline scheme: The additional structures needed to support exclusive squashing will be discussed in Section 6.

Fig. 1(a) shows the data structures defined by the original scheme. This scheme assumes that all the data that will be speculative accessed is packed inside a "reference vector" (**ref** in the figure). This solution allows to a fast implementation of speculative loads, stores, and commit operations. However, this advantage comes at the cost of the need of packing/unpacking speculative data before the parallel execution if speculative variables are not into a single data structure. The additional data structures needed by the original speculative parallelization scheme are the following:

4.1 Window

This is a vector of W slots used to implement the sliding-window mechanism. W represents the window size used, typically a small multiple of the number of threads available. Elements in this window store the status of different threads associated to each slot. When a thread starts the computation of a new block of iterations, a new window slot after the most speculative one is assigned and its state is set to "RUNNING". Other possible states are "DONE", indicating that the thread has already finish with its chunk of iterations and the results should be committed; "SQUASHED", indicating that the associated thread has been squashed, or "FREE" if the window is not being used. Two variables, called *non_spec* and *most_spec*, indicate which slots are used by the threads that execute, respectively, the non-speculative and most-speculative chunks.

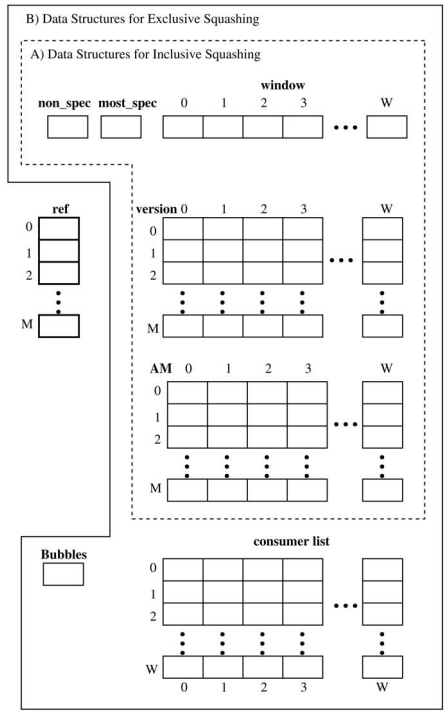

Fig. 1. Data structures needed for the (a) inclusive and (b) exclusive versions of the speculative scheme used.

4.2 Version

This is a $M \times W$ matrix used to store the version copies of the M elements that form the data structure being speculatively accessed. The main copy of this structure is called *ref* in the figure. A thread running in slot i is able to modify its version copy (column i of the Version array), and to read version copies owned by other threads. Although this solution is not space-efficient when not all threads access all speculative data, it allows to a very fast forwarding of data from predecessors, simply by requesting the corresponding datum. It also allows to an equally fast checking for dependence violations.

4.3 AM (Access Matrix)

This is a $M \times W$ matrix used to store the state of each corresponding value stored in the *Version* matrix. State values include NotAcc, indicating that the element has not been used yet by the current thread; ExpLd, indicating that the current thread has forwarded a value for this element from a predecessor; ExpLdMod, indicating that the current thread has forwarded and later modified the value for this element; and Mod, indicating that the current thread has written a new value for this element. There are other states that handle reduction operations, not shown here for simplicity.

We have optimized the original speculative parallelization scheme with the use of W **MDL (Modified Data Lists)** (not shown in Fig. 1 for simplicity). For each window slot, we maintain a list of the indices of the data that has been modified by the thread working in each window slot. The list is updated after every speculative store. Its use leads to a faster commit operation, since the non-speculative thread does not longer need to traverse the entire Access Matrix in order to find the elements that should be committed to the reference copy.

5 Inclusive Squashing Implementation

As we stated in Section 2, a speculative parallelization scheme should perform several operations, including speculative loads, speculative stores, and commit and thread management. We will now briefly describe how the base scheme handles each operation, and then we will show the changes needed to implement the exclusive squash mechanism.

5.1 Inclusive Speculative Loads

Algorithm 1 shows the speculative load process in detail. If a thread is using slot *tid* of the data structure and it has never accessed this value (line 1), it first marks the value as speculatively loaded (line 2), and ensures that all threads flush their cache copies of the *AM* matrix, being aware of this state change (line 3). This memory fence also ensures that the compiler will not reorder the associated instructions, getting the value before setting its state, that would lead to a race condition. Our thread now searches backwards in the *AM* structure for a copy of the value in states Mod or ExpLdMod (line 5), and forwards this value to its own version copy (line 6). If it has not found any predecessor with a value for this element, it gets the reference version (line 11).

It is interesting to note that values in ExpLd state should not be forwarded. Since our thread first marks the value to be read as ExpLd in the *AM* column, and later forwards the value, a thread that is more speculative than ours might see our state as ExpLd and try to forward the value before our thread has time to complete the speculative load.

Algorithm 1. Inclusive speculative load operation

input: Main copy of the speculative data (*ref*), index of the speculative element to be accessed i, index of current thread *tid*

output: Most up-to-date version of i-th speculative element is returned.

```
1  if AM[i][tid] = NotAcc then
2      AM[i][tid] ← ExpLd
3      #pragma memory fence
4      for j ← tid − 1 to non_spec do
5          if (AM[i][j] = Mod) or
           (AM[i][j] = ExpLdMod) then
6              version[i][tid] ← version[i][j]
7              forwarded ← TRUE
8              break
9          end
10     end
11     if forwarded ≠ TRUE then
       version[i][tid] ← ref[i]
12 end
13 Return version[i][tid]
```

5.2 Inclusive Speculative Stores

If a thread should write a speculative value, it stores the value in its own version copy, and sets its state to *ExpLdMod* or *Mod* as needed. If an eager squashing policy is used, this speculative store also should search for threads that could have consumed an outdated version of this value. If a lazy squashing policy is used, this search is postponed until the non-speculative thread starts a commit operation. The search operation to be carried out in both cases is described below.

5.3 Inclusive Search for Dependence Violations

The inclusive search for violations can be carried out by each speculative store operation (eager version) or by the non-speculative thread before committing data (lazy version). In both cases, a forward search looks for any successor that have used an outdated copy of the value (looking for entries in ExpLd or ExpLdMod states in the *AM* matrix). If such a successor is found, our thread triggers a squash operation, marking the first offending thread and all its successors as SQUASHED. Algorithm 2 shows the search for dependence violations in detail. If thread *tid* finds that thread j has consumed a wrong value, a dependence violation is found (lines 1–3). In this case, the offending thread and all its successors are squashed inside a critical section (lines 4–10), marking thread $j-1$ as the new most-speculative thread. Since thread j and all its successors have been squashed, our thread stops the search for violations (line 11).

Algorithm 2. Inclusive search for violations and squash mechanism

input: Index of the speculative element to be accessed i, index of current thread *tid*.

output: Successors that have consumed a wrong value are squashed, together with **all** their successors.

```
1  for j ← tid + 1 to most_spec do
2      if AM[j][tid] = Mod then break; // shield found
3      else if AM[j][tid] ≠ NotAcc then
4          #pragma critical section
5          for k ← j to most_spec do
6              if window[k] = RUNNING then
                   window[k] ← SQUASHED
7              else if window[k] = DONE then
                   window[k] ← FREE
8          end
9          most_spec ← (j-1)
10         # pragma end critical section
11         break
12     end
13 end
```

5.4 Inclusive Commit and Thread Management

When a thread finishes the execution of a chunk of iterations, it enters in a critical section to commit results. Inside this critical section, one of the following three situations can arise:

1. The thread is found to be the non-speculative one. In this case, the thread should commit all its changes to the *ref* version, and also commit all the data generated by all consecutive successors to it that are in the DONE state. After this commit operation, the *non-spec* pointer is advanced to the first thread not yet committed, and that becomes the new non-speculative one. If the eager inclusive squashing policy is used, no search for violations is carried out at this moment. However, the use of the lazy inclusive squashing policy forces the non-speculative thread to search for possible dependence violations (using Algorithm 2) before committing the data generated by each thread.
2. The thread is found to be speculative. In this case, it only changes its own state from RUNNING to DONE, letting the non-speculative thread to later commit its results.
3. The thread has been squashed. In this case, its slot state is changed to FREE and no further action is required.

After that, the thread exits the critical section, and if the sliding window is full, it will spin-wait until the sliding window is moved by the non-speculative thread, thus generating free slots. The thread then enters the critical section, acquires the next available slot and its associated chunk of iterations to be computed, increments the *most-spec* pointer, exits the critical section and starts the work. Note that this thread will always be the new most-speculative thread. This behavior ensures that all window elements between the *non_spec* pointer and the *most_spec* pointer are either RUNNING or DONE, keeping the sliding window free of squashed threads.

Algorithm 3. Speculative load operation, exclusive version

input: Main copy of the speculative data (*ref*), index of the speculative element to be accessed i, index of current thread *tid*.

output: Most up-to-date version of i-th speculative element is returned.

```
1  if AM[i][tid] = NotAcc then
2      AM[i][tid] ← ExpLd
3      #pragma memory fence
4      for j ← tid − 1 to non_spec do
5          if (window[j] = DONE) or
           (window[j] = RUNNING) then
6              if (AM[j][tid] = Mod) or
               (AM[j][tid] = ExpLdMod) then
7                  #pragma memory fence
8                  consumer_list[tid][j] ← TRUE
9                  if (window[j] = DONE) or
                   (window[j] = RUNNING) then
10                     version[i][tid] ← version[i][j]
```

```
11          forwarded ← TRUE
12          break
13        end
14      end
15    end
16  end
17  if forwarded ≠ TRUE then version[i][tid] ← ref[i]
18 end
19 Return version[i][tid]
```

6 Exclusive Squashing Implementation

After reviewing the general behavior of the speculative scheme used as a baseline for our study, we will see how its operations should be modified to handle both eager and lazy exclusive squashing. The design goals of our system are:

- Squashing only *offending threads*, together with any threads that may consumed incorrect data from the offending threads (called *consumer threads*).
- Avoiding the use of additional critical sections to handle speculative loads and stores, since these operations are so common that the need of critical sections would destroy any performance gain.

The exclusive squash mechanism proposed extends the data structures used in the speculative scheme with a new *consumer_list* matrix (see Fig. 1(b)). This structure is a boolean $W \times W$ matrix that keeps the relationship between each thread and all successor threads that may have consumed a value from it. If thread i has loaded a speculative datum previously stored by thread j, *consumer_list[i][j]* is set to TRUE. If thread j has consumed an outdated version of a datum, the offending thread j and also all threads marked as consumers in column j of the *consumer_list* matrix will be squashed. All consumers of data stored in squashed threads are squashed as well. As we will see, this structure is enough to effectively track inter-thread dependencies.

All the basic operations described in the previous section (speculative loads, speculative stores, and thread management) need the following changes to handle this new squashing policy.

6.1 Exclusive Speculative Loads

Recall that the speculative load operation should forward the most up-to-date value from a predecessor thread, returning the reference value if no predecessor has used the value so far. To implement the exclusive squash mechanism, we should take into account two different issues. First, the new speculative load operation should keep record of the data dependence generated by the forwarding operation, by writing into the *consumer_list* structure. Second, when searching for the most up-to-date version of the datum, the speculative load should be aware that now there may be squashed or empty slots between our thread and the non-speculative one. These "bubbles" are a consequence of selectively squashing threads.

We have found a way to perform both operations without the use of any additional critical section. Algorithm 3 shows the exclusive load speculative operation. The main differences between the inclusive version and this new version are in lines 5 and 7–9. Line 5 checks if the predecessor being examined is still valid (in RUNNING or DONE states). If the predecessor is valid and it has the datum, our thread proceeds with the forwarding. After a memory fence to avoid reordering (line 7), the new dependence generated is stored (line 8). Before effectively perform the forwarding (lines 10–12), our thread checks again that the predecessor is still alive (line 9). This check avoids the following race condition: If our thread just checks that the predecessor is alive (line 5), store the dependence (line 8) and get the value (line 10), the predecessor might be squashed while our thread is executing lines 5 to 8, and the new dependence would remain unnoticed. The additional check in line 9 ensures that our thread will skip that value in this case. This race condition does not exist in the base system, because if the predecessor is squashed, our thread will be squashed as well.

6.2 Exclusive Speculative Stores

With respect to the storage of the new value, speculative stores are the same regardless of the squashing mechanism used, since accounting for inter-thread dependences are carried out in speculative load operations.

Algorithm 4. Exclusive search for violations and squash mechanism

input: Index of the speculative element to be accessed i, index of current thread *tid*.

output: Successors that have consumed a wrong value are squashed, together with all their successors that have consumed a value from them.

```
1 for j ← tid + 1 to most_spec do
2   if (window[j] ≠ SQUASHED) and (window[j] ≠ FREE) then
3     if AM[j][tid] = Mod then break; // shield found
4     else if (AM[j][tid] ≠ NotAcc) then
5       # pragma critical section
6       if (window[j] = RUNNING) then window[j] ← SQUASHED
7       else if (window[j] = DONE) then
8         window[j] ← FREE
```

```
9                 bubbles ← bubbles + 1
10              end
11              for k = j + 1 to most_spec do // Search for additional offending threads
12                if (window[k] ≠ SQUASHED) and (window[k] ≠ FREE) then
13                  if (AM[k][tid] = Mod) then break (shield found)
14                  else if (AM[k][tid] ≠ NotAcc) then
15                    if (window[k] = RUNNING) then window[k] ← SQUASHED
16                    else if (window[k] = DONE) then
17                      window[k] ← FREE
18                      bubbles ← bubbles + 1
19                    end
20                  end
21                end
22              end
23              for k = j to most_spec do // Search for consumers
24                if (window[k] = SQUASHED) or (window[k] = FREE) then
25                  for c = (k + 1) to most_spec do
26                    if (consumer_list[c][k] = TRUE) then
27                      if (window[c] = RUNNING) then
28                        window[c] ← SQUASHED
29                      end
30                      else if (window[c] = DONE) then
31                        window[c] ← FREE
32                        bubbles ← bubbles + 1
33                      end
34                    end
35                  end
36                end
37              end
38              # pragma end critical section
39              break
40            end
41          end
42        end
```

6.3 Exclusive Search for Violations

This is the core part of the exclusive squashing mechanism. As with the inclusive squash policy, the search for violations can be carried out after each speculative store (eager version) or at commit time (lazy version). While the inclusive squash policy searches for *the first* offending thread and then squashes that thread and all its successors, the exclusive squash mechanism proceeds in two phases. First, it will selectively squash *every* offending thread. Second, it will squash all threads that have consumed *any value* from a squashed thread. Since our system just store inter-thread data dependencies, this operation squashes all consumers even if the data forwarded is not related to the datum that has triggered the squash operation.

Algorithm 4 shows the exclusive search for violations and squashing mechanism. When searching for offending threads, our thread *tid* should skip window slots that are either in

SQUASHED or FREE states (line 2). If it finds an offending thread (line 4), it will enter in a critical section (line 5). Note that no critical section will be entered if no dependencies arise, an important issue to guarantee a fast execution in this case. Inside this critical section it marks the offending thread as SQUASHED or FREE (lines 6–10), and proceed in the same way with any additional offending threads (lines 11–22). Afterwards, our thread will examine the consumer list associated with the squashed threads, squashing them as well (lines 23–37). Since squashes are performed selectively, the *most_spec* pointer is not changed here but in the thread management function, described below.

6.4 New Commit and Thread Management

The commit operation in the exclusive squash mechanism does not change in comparison with the inclusive mechanism described in the previous section. Regarding thread management, the main difference between the inclusive and exclusive version is the management of "bubbles" of SQUASHED or FREE slots inside the sliding window. Recall that the inclusive thread management operation always assigns new threads for execution by advancing the *most_spec* pointer. In our scheme, there may be some FREE cells in the window that should be assigned for re-execution before advancing the *most_spec* pointer. A new global variable, called *bubbles*, records the number of FREE cells in the window. This variable is incremented each time a window cell reaches the FREE state due to a dependence violation (lines 9, 18 and 32 of the speculative store operation shown in Alg. 4). If *bubbles* is not zero, the thread management function will search for the FREE cell, assigning it for execution to the current thread, and decreasing the counter. Otherwise, the *most_spec* pointer is advanced as usual.

An execution example that describes this process in more detail can be found in the Supplementary Material, which can be found in the Computer Society Digital Library at https://doi.ieeecomputersociety.org/10.1109/TC.2013.46.

7 Tradeoffs and Cost Analysis

As we have shown in the previous section, the implementation of this exclusive squashing mechanism requires some changes in the speculative load and store operations and in the thread management functions. These changes result in unnecessary operations if the loop speculatively parallelized has no dependence violations and no squashes are needed. Fortunately, these costs are indeed modest. A detailed discussion follows:

- Speculative loads: To implement the new mechanism, the only additional operations are to skip empty slots when searching for the most up-to-date values, and to mark the new dependence in the *consumer_list* structure. Although these operations are not costly by themselves, they are done in every speculative load, adversely affecting performance. The asymptotical cost of the speculative load operation is the same than in the inclusive squashing version.
- Speculative stores: The search for violations in both schemes are the same, so the new solution does not affect performance if no squashes arise. In the case of a dependence violation, the squash procedure inside the critical section is in $O(P^2)$ instead of $O(P)$, being P the number of threads. Depending the application, the benefits of avoiding unnecessary squashes may compensate for this cost.
- Thread management: In the new scheme, a finishing thread will first look if there are any "bubbles" in the sliding window and choose the associated slot in that case. The cost of this additional operation is in $O(P)$, and should be performed only if a bubble exists.

It is important to highlight that our implementation of the exclusive squashing mechanism does not add any additional critical section to the original solution. Therefore, scalability is not compromised, despite of the extra work carried out inside the critical sections. Finally, it is worthwhile to note that this solution allows a very flexible implementation of the speculative scheme used, since the squash policy can be changed in any moment, even *during* the speculative execution of a loop. To do so, the scheme just needs to squash all running threads and re-scheduling them with a different squashing policy.

8 Experimental Results

Experiments were carried out on an Intel S7000FC4URE server, equipped with four quad-core Intel Xeon MPE7310 processors at 1.6 GHz and 32 GB 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 the `gcc` compiler suite for C applications, with `-O4` optimization level, and SunStudio f90 compiler for Fortran applications, with `-O4 -xarch=native` optimization flags. 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.

8.1 Applications Considered

To evaluate our new scheme, we compare the behavior of an implementation of the speculative scheme described in [1] with a modified version that incorporates our exclusive squashing mechanism. We evaluate our solution with six different applications, both with and without dependence violations produced at runtime.

Three of the applications considered are written in Fortran, and three are written in C. To parallelize these applications, we have developed C and Fortran versions of the speculative parallelization schemes being evaluated, using OpenMP [19] task-parallelism primitives.

8.2 Loops With No Dependence Violations

The first part of our study shows how the use of different squashing mechanisms affects the performance of applications whose loops do not produce dependence violations. We study three Fortran applications: TREE [20], MDG (part of the PERFECT Club Benchmark suite [21]), and SPECfp2000's WUPWISE [22]. Each one of these applications have one or more loops (`accel_10` in TREE, `muldeo_200` and `muldoe_200` in MDG, and `interf_1000` in WUPWISE) that consume a significant part of the execution time, but that cannot be parallelized at compile time because they access data structures using indices whose values depend on the program control flow and/or the data input sets. Despite this fact, these loops do not

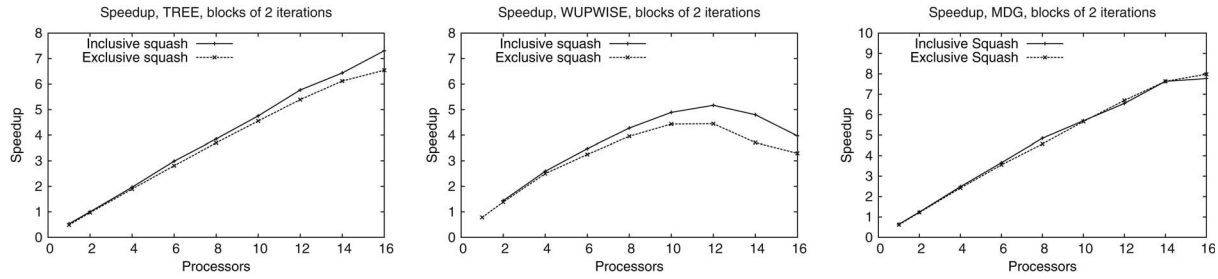

Fig. 2. Speedup results for applications with no dependence violations.

produce any dependence violations at runtime, making them excellent candidates to be parallelized using speculative parallelization techniques.

Fig. 2 shows the relative performance of TREE, MDG and WUPWISE chosen loops when executed with our software-based speculation scheme and both squash mechanisms described before: Inclusive and Exclusive Squash. Since no dependence violations arise at runtime, both eager and lazy solutions lead to the same performance results.

The results show a slowdown ranging from 2.7% for MDG to 16.2% for WUPWISE. As we stated in Section 7, this performance degradation of the exclusive squash is due to the cost of keeping data dependencies among threads in the speculative load operation. As far as these applications present no dependence violations, no performance gains alleviate this cost. Note that speculative stores do not contribute to the performance degradation, since their behavior is the same for both versions due to the lack of runtime dependence violations.

As we suggested in Section 7, the runtime system might use the inclusive squashing policy by default, and switch on-the-fly to the exclusive squashing mechanism if the number of dependence violations that arise exceeds a certain threshold.

8.3 Influence of Dependence Violation Rates

After measuring the effects of the squashing mechanisms in applications whose loops do not produce dependence violations at runtime, we will now study how the behavior of these mechanisms is affected by different rates of dependence violation in the same application. To do so, we will use a C implementation of the randomized incremental algorithm that builds the Convex Hull of a two-dimensional set of points. This algorithm, that we will call 2D-Hull, due to Clarkson et al. [23], computes the convex hull (smallest enclosing polygon) of a set of points in the plane. The input to Clarkson's algorithm is a set of (x, y) point coordinates. 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.

The probability of a dependence violation in the 2D-Hull algorithm depends on the shape of the input set. For example, if N points are distributed uniformly on a disk, the i-th iteration will present a dependence with probability in $\theta(\sqrt{i}/i)$. If points lie uniformly on a square, the probability of a dependence will be in $\theta(\log(i)/i)$.

We have compared the performance of our different squashing mechanisms using four different, 10-million point input sets. The first one, Kuzmin, is an input set that follows a Gauss-Kuzmin distribution, where the density of points is higher around the center of the distribution space. This input set leads to very few dependence violations, since points far from the center are very scarce. The Square and Disc input sets are uniform distributions of points inside a square and a disc, respectively. It is easy to see that the Square input set leads to an enclosing polygon with fewer edges than the Disc input set, thus generating fewer dependence violations. Finally, the Circle input set distributes all the points around a circle, leading to a huge number of dependence violations.

The effects of these input sets in the execution of the 2D-Hull are summarized in Table 1. Both the amount of data speculatively accessed and the number of dependence violations depend on the number of edges in the final convex hull.

Fig. 3 shows the results obtained in the execution of the 2D Hull for these input sets. Results include the speedup obtained, the number of dependence violations that have arisen, and the total number of the squashed threads for the four squashing alternatives examined. Each plot indicates the size of the blocks of consecutive iterations scheduled to each thread.

From the results we can draw the following observations. First, the eager exclusive squash policy leads to a relative performance improvement up to 1.9% for Kuzmin, 9.0% for Square, 14.2% for Disc, and 5.7% for Circle input sets with respect to the eager inclusive policy. As expected, the higher the rate of dependence violations, the more the benefits of this new squashing policy technique. However, the extremely high number of dependence violations with the Circle input set makes squashes so frequent that the benefits derived from the exclusive squashing policy are subsumed by a 3x performance slowdown.

Second, eager versions (that search for dependence violations in every speculative store operation) leads to better speedup results in all cases. This situation, that is consistent

TABLE 1
Characteristics and Effects of Different Input Sets When Computing the 2D-Hull Algorithm

Input set	Edges of the convex hull	Spec data size per thread	Dependence violations
Kuzmin	161	4 Kb	0.0008 %
Square	647	15.25 Kb	0.0032 %
Disc	4 393	103.14 Kb	0.0219 %
Circle	206 797	4.73 Mb	10.3400 %

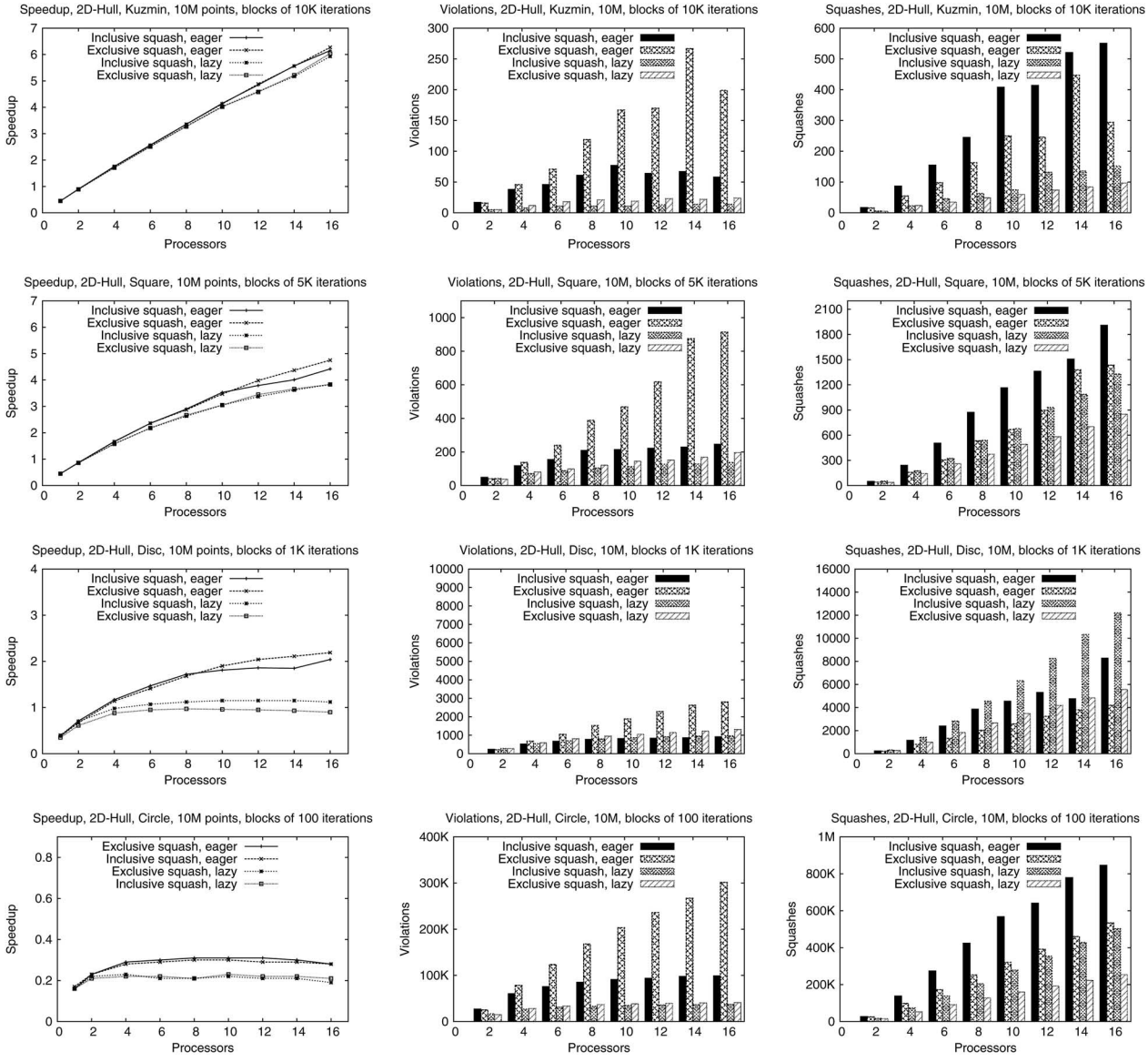

Fig. 3. Speedup, violations and squashes generated by the four versions of the speculative engine while computing the two-dimensional Convex Hull of 10 M points using different input sets.

with our earlier study [14], is motivated by the added contention time due to the longer check-and-commit operation. In lazy versions, the search for violations is inside the critical section, thus belonging to the critical execution path. On the other hand, in eager versions searches are carried out concurrently by all threads, only entering the critical section when a dependence violation is found.

Third, regarding dependence violations and number of squashed threads, it is interesting to note that the exclusive squashing mechanism leads to twice dependence violations on average, although only around two thirds of the threads are squashed. Regarding eager versions, the results show a relative dependence violation increment of 223.58% for Kuzmin, 224.03% for Square, 210.86% for Disc, and 204.70% for Circle input sets on average. On the contrary, the number of squashed threads decrease by 31.63% for Kuzmin, 29.92% for Square, 35.46% for Disc, and 34.07% for Circle input sets on average. The rationale of this effect is the following. For each dependence violation found, inclusive mechanisms squash all successors, preventing the occurrence of new dependence violations that are "waiting to happen" since *all successors* will sooner or later use the speculative solution to check whether their points lie outside the convex hull. On the contrary, exclusive squash mechanisms only squash offending threads and their current consumers, but the survivals will generate new dependence violations that lead to additional, exclusive squash operations. As the experimental results show, this situation does not depend on the particular rate of dependence violations.

We can conclude that, despite the increment of the number of dependence violations triggered, the use of the exclusive squashing policies leads to noticeable performance improvement in all cases.

8.4 Influence of the Speculative Data Access Pattern

The last part of our study examines the influence of the squashing mechanism in the execution of two C applications with very different access patterns to the speculative data.

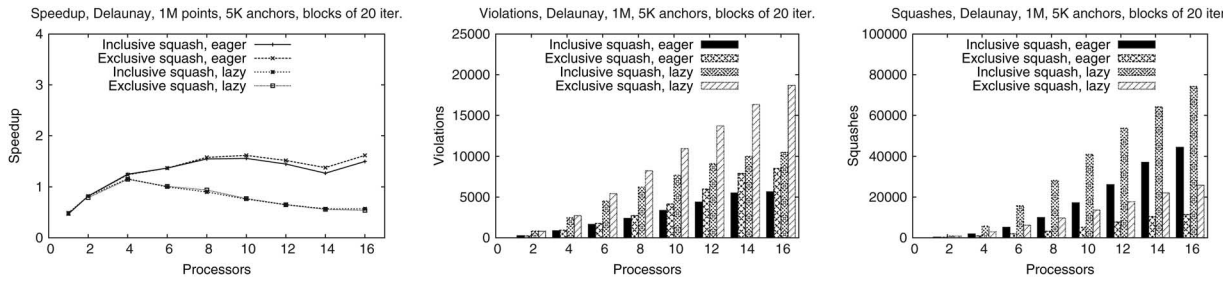

Fig. 4. Speedup, violations and squashes generated by the four versions of the speculative engine while computing the two-dimensional Delaunay triangulation.

8.4.1 The Delaunay Triangulation

The first application is the randomized incremental construction of the Delaunay triangulation using the *Jump-and-Walk* strategy, introduced by Mücke et al. [24], [25]. This incremental strategy starts with a number of points, called anchors, whose containing triangles are known. The algorithm finds the closest anchor to the point to be inserted (the *jump* phase), and then traverses the current triangulation until the triangle that contains the point to be inserted is found (the *walk* phase). After this location step, the algorithm divides this triangle into three new triangles, and then updates the surrounding edges to keep the Delaunay properties. This local modification to the current Delaunay solution may lead to dependence violations, since other threads may have traversed the old solution while trying to add new points.

The shared data access pattern of the Delaunay algorithm is fundamentally different than the one of the 2D-Hull. In the Delaunay algorithm all threads modify the speculative solution, not only a subset of them. However, despite these modifications, successor threads do not need to be squashed if their own work is carried out in a different zone of the triangulation.

The expected amount of dependence violations generated by the Delaunay Triangulation depends on the number of processors and the length of the traversing path. It is easy to see that, the shorter the distance between the closest anchor and the point to be inserted, the fewer triangles that are visited in the walk and the smaller the probability of a dependence violation. This fact suggests that the algorithm should work with many anchors. However, the bigger the number of anchors, the more distance comparisons have to be performed to find the closest anchor to our point, thus degrading sequential performance. Our implementation uses a number of anchors that represents a good balance between these effects for the input size used. Our implementation is composed by two loops: The first one builds a Delaunay Triangulation of the first 5 000 points, that will be used later as anchors, while the second loop inserts all the remaining points (up to one million). We have speculatively parallelized this second loop.

Fig. 4 shows the speedup, number of dependence violations and squashes produced during the execution of the Delaunay Triangulation algorithm on a set of one million points uniformly distributed on a disc. The performance results are consistent with those obtained with the 2D-Hull algorithm, with a relative speedup increment of 8.90% for the eager exclusive squashing policy. As in the 2D-Hull case, the number of dependence violations generated by the eager squashing policy is higher (122.33% on average), but the number of squashed threads decreases by 58% on average.

It is interesting to examine the reasons for the poor performance results of the lazy squashing policies in this case. Since in this algorithm all threads modify the solution, all window slots have data to be committed by the non-speculative thread. Therefore, the additional cost due to the search for dependence violations at commit time is proportionally higher in this case than in the 2D-Hull algorithm, leading to a significant performance slowdown.

8.4.2 The Minimum Enclosing Circle

The 2-dimensional Minimum Enclosing Circle (2D-MEC) is the smallest circle that comprises a set of points. We will study the behavior of the randomized incremental construction due to Welzl [26]. Due to its randomized incremental nature, the algorithm solves the problem in linear time. This algorithm starts with a circle of radius equal to zero located in the center of the search space. If a point lies outside the current solution, the algorithm defines a new circle that uses this point as one of their frontiers. It is interesting to note that points that laid inside the old solution may laid outside the new one. Therefore, all points should be processed again to check if the new circle encloses them. The solution is defined by two or three points, and the algorithm is composed of three nested loops. We have speculatively parallelized the innermost loop, that consumes 45% of the total execution time.

The speculative data access pattern of this application is different than the access pattern of the 2D-Hull and Delaunay algorithms. In this case, the data shared among threads are the coordinates of two or three points that define the current solution. If this solution changes, *all* successors should be squashed, making exclusive squashing policies less useful in this case.

Fig. 5 shows the speedup, number of violations and squashes produced by the four combinations of squashing policies being studied. As expected, the lazy inclusive squash leads to the best performance for this application. With respect to exclusive policies, the eager exclusive squash presents a performance degradation when augmenting the number of threads. The reason is that this policy combines a higher number of dependence violations (since the mechanism does not avoid dependence violations that will happen shortly) with a higher number of squashes (due to the high number of successors affected by each dependence violation). Lazy versions, on the other hand, minimize the number of dependence violations, leading to better performance.

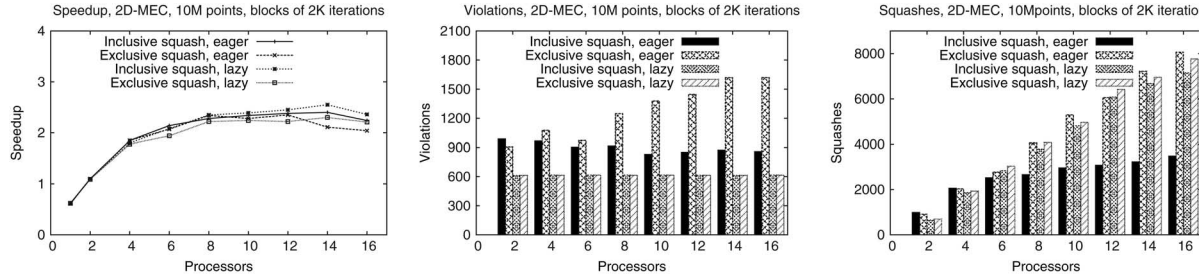

Fig. 5. Speedup, violations and squashes generated by the four versions of the speculative engine while computing the two-dimensional Minimum Enclosing Circle.

A more detailed discussion on the applicability of both squashing alternatives can be found in the Supplementary Material, available online.

9 Related Work

Since squashes are unavoidable if an inter-thread dependence violation occurs, a great effort has been done in reducing their occurrences. However, to the best of our knowledge, no software-based speculation scheme with exclusive squashing policy has been proposed so far in the literature.

Regarding hardware TLS systems, a combination of techniques are used in [27] to reduce the number of squashes produced by a hardware speculative parallelization mechanism with inclusive squashing. An analysis of approaches to buffer and manage multi-version speculative memory state in multiprocessors is presented in [28], together with a detailed tradeoff analysis for both single and multi-chip multiprocessors. The study is also based on the use of inclusive squashing. A hardware architecture for TLS in high-coupled multiprocessors is presented in [29]. The execution mechanism is composed by two threads: one non-speculative and other speculative. Once the non-speculative thread reaches the starting point of the speculative thread, it replays all incorrect instructions with the help of a hardware speculation result buffer. Thanks to the hardware support and to the fact that only two threads cooperate in the task, squashes are always "perfect", in the sense described in Section 3. In [13], a model that combines different techniques such as thread-level speculation, helper threads and run-ahead execution is proposed to dynamically choose at runtime the most appropriate combination. The proposed TLS system also uses inclusive squashes. An adaptive approach for speculative loop execution that handles nested loops has been recently proposed [30]. This proposal also relies on inclusive squash to squash the offending thread and all its successors at the same nesting level.

Several works propose speculative parallelization mechanisms that benefit from different degrees of code transformations. Tian et al. [31] propose the use of the Copy-or-Discard (CorD) execution model to avoid expensive state recovering mechanisms in case of misspeculation. This proposal requires an in-depth analysis of the original loop, and the use of code transformation techniques that reduce the probability of misspeculation. The mechanism proposed just squashes the offending thread and not its successors, simply because this proposal does not use forwarding to recover the most up-to-date value of speculative variables. Instead, speculative loads in this proposal always get the non-speculative version of the data, so successors of the offending thread are not affected by misspeculations. This design tradeoff leads to good speedups when dependence violations are highly unlikely. This is the case for the CorD execution model, because it minimizes the possibility of dependence violations thanks to its compile-time code transformations.

A recent paper of the CorD group [32] also aims to reduce the cost of misspeculation, but from a different perspective. Instead of squashing only offending threads and other threads that have used their data, this proposal records intermediate states during the speculative execution. In this way, instead of aborting a complete task, only a portion of the task is re-executed. This solution comes at the cost of a more complex code analysis, in order to insert intermediate checkpoints where the earliest reads of the speculative variables are found.

In [17], a software-based TLS system is proposed to help in the manual parallelization of applications. The system requires from the programmer to mark "possibly parallel regions" (PPR) in the application to be parallelized. The system relies on a so-called "tournament" model, with different thread cooperating to execute the region speculatively, while an additional thread runs the same code sequentially. If a single dependence arises, speculation fails entirely and the sequential execution results is used instead. The usefulness of this system is based on the assumption that the code chosen by the programmer will likely not present any dependencies. An improvement to this scheme is described in [33], relying on dependence hints provided by the programmer to allow explicit data communication between threads, thus reducing runtime dependence violations.

Finally, exclusive squashing at a finer lever of granularity has been proposed in the context of transactional memory. Ramadan et al. [34] perform value forwarding while tracking individual dependences between transactions, allowing to only abort dependent transactions. Since the system proposed, called DASTM, is based in the use of effective addresses, it works at a finer level of granularity than SP, and therefore there is almost no possibility of discarding potentially valid work due to false-sharing issues. However, in the context of speculative parallelization, different threads may access to different data in the same speculative working set. This coarser level of granularity makes false-sharing situations common. As far as we know, our solution is the first system that avoid losses of valid work in this case.

10 Conclusions

This work explores the design space of solutions to avoid unnecessary squashes in software-based speculative parallelization, and their evaluation in terms of performance. In the event of a dependence violation, most software-based TLS systems discard the work done by the offending thread and all its successors, potentially losing and repeating correct computations. In this work we devise a new squashing policy, called "exclusive squashing", that only squashes threads that have effectively consumed an incorrect value, and all their successors that have consumed any value from them. We have performed an exhaustive exploration of the squashing policy design space, not only with respect to the squashing strategy being used, but also taking into account the different stages when dependence violations can be detected and corrected. Our experimental results show that the exclusive squashing policy effectively leads to performance improvements in applications where a dependence violation does not invalidate all the work carried out by the successors of the offending thread.

The speculative parallelization scheme and benchmarks described in this paper are available under request.

Acknowledgments

The authors would like to thank the anonymous referees for their valuable suggestions, and Dr. Marcelo Cintra and Dr. Belén Palop for many helpful discussions on this topic. This research is partly supported by the Spanish Government (TIN2007-62302, TSI-020302-2008-89, CENIT OCEANLIDER), the CAPAP-H network (TIN2010-12011-E, TIN2011-15734-E), and Junta de Castilla y León, Spain (VA172A12-2). Part of this work was carried out under the HPC-EUROPA2 project (project number: 228398), with the support of the European Community, Research Infrastructure Action of the FP7.

References

[1] M. Cintra and D. R. Llanos, "Toward efficient and robust software speculative parallelization on multiprocessors," in *Proc. 9th ACM SIGPLAN Symp. Principles Practice Parallel Program. (PPoPP)*, 2003, pp. 13–24.

[2] F. H. Dang, H. Yu, and L. Rauchwerger, "The R-LRPD test: Speculative parallelization of partially parallel loops," in *Proc. 16th Parallel Distrib. Process. Symp.*, 2002, pp. 20–29.

[3] M. Gupta and R. Nim, "Techniques for speculative run-time parallelization of loops," in *Proc. IEEE/ACM Conf. Supercomput. (ICS)*, 1998, pp. 1–12.

[4] M. Kulkarni et al., "Optimistic parallelism benefits from data partitioning," in *Proc. 13th Architectural Support Program. Languages Operat. Syst. (ASPLOS)*, 2008, pp. 233–243.

[5] M. Kulkarni et al., "Optimistic parallelism requires abstractions," in *Proc. Program. Language Des. Implemen. (PLDI)*, 2007, pp. 211–222.

[6] M. Cintra, J. F. Marítnez, and J. Torrellas, "Architectural support for scalable speculative parallelization in shared-memory multiprocessors," in *Proc. 27th Int. Symp. Comput. Archit. (ISCA)*, 2000, pp. 13–24.

[7] L. Hammond, M. Willey, and K. Olukotun, "Data speculation support for a chip multiprocessor," in *Proc. 8th Int. Conf. Architectural Support Programm. Languages Operat. Syst.*, 1998, pp. 58–69.

[8] P. Marcuello and A. González, "Clustered speculative multithreaded processors," in *Proc. 13th Int. Conf. Supercomput. (ICS)*, 1999, pp. 365–372.

[9] J. G. Steffan et al., "A scalable approach to thread-level speculation," in *Proc. 27th Annu. Int. Symp. Comput. Archit.*, 2000, pp. 1–12.

[10] L. R. Y. Zhan and J. Torrellas, "Hardware for speculative Run-Time parallelization in distributed Shared-Memory multiprocessors," in *Proc. 4th Int. Symp. High-Perform. Comput. Archit. (HPCA)*, 1998, p. 162.

[11] L. Rauchwerger and D. Padua, "The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization," in *Proc. Conf. Programm. Lang. Des. Implement.*, pp. 218–232, Jun. 1995.

[12] P. Rundberg and P. Stenström, "An all-Software thread-level data dependence speculation system for multiprocessors," *J. Instr.-Level Parallelism*, vol. 3, Oct. 2001.

[13] P. Xekalakis, N. Ioannou, and M. Cintra, "Combining thread level speculation helper threads and runahead execution," in *Proc. 23rd Int. Conf. Supercomput. (ICS)*, 2009, pp. 410–420.

[14] M. Cintra and D. R. Llanos, "Design space exploration of a software speculative parallelization scheme," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 6, pp. 562–576, Jun. 2005.

[15] C. Kruskal and A. Weiss, "Allocating independent subtasks on parallel processors," *IEEE Trans. Softw. Eng.*, vol. SE-11, no. 10, pp. 1001–1016, Oct. 1985.

[16] M. Kulkarni et al., "Scheduling strategies for optimistic parallel execution of irregular programs," in *Proc. 20th Annu. Symp. Parallelism Algorithms Archit. (SPAA)*, 2008, pp. 217–228.

[17] C. Ding et al., "Software behavior oriented parallelization," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation (PLDI)*, 2007, pp. 223–234.

[18] J. G. Steffan et al., "The STAMPede approach to thread-level speculation," *ACM Trans. Comput. Syst.*, vol. 23, no. 3, pp. 253–300, 2005.

[19] R. Chandra et al., *Parallel Programming in OpenMP*, 1st ed. San Mateo, CA: Morgan Kaufmann, Oct. 2000.

[20] J. E. Barnes. (Jan. 1997). *TREE*. Hawaii: Institute for Astronomy, University of Hawaii. [Online]. Available: ftp://ftp.ifa.hawaii.edu/pub/barnes/treecode/

[21] M. Berry, D. Chen, and P. Koss, "The PERFECT club benchmarks: Effective performance evaluation of supercomputers," *Int. J. Supercomput. Appl.*, vol. 3, no. 3, pp. 5–40, 1989.

[22] Standard Performance Evaluation Council, *SPEC CPU2000 benchmark suite* [online]. Available: http://www.spec.org/cpu2000/

[23] K. L. Clarkson, K. Mehlhorn, and R. Seidel, "Four results on randomized incremental constructions," *Comput. Geom. Theory Appl.*, vol. 3, no. 4, pp. 185–212, 1993.

[24] L. Devroye, E. P. Mücke, and B. Zhu, "A note on point location in Delaunay triangulations of random points," *Algorithmica*, vol. 22, pp. 477–482, 1998.

[25] E. P. Mücke, I. Saias, and B. Zhu, "Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations," in *Proc. 12th ACM Symp. Comput. Geom.*, 1996, pp. 274–283.

[26] E. Welzl, "Smallest enclosing disks (balls and ellipsoids)," in *New Results New Trends Comput. Sci.* (Lecture Notes Comput. Sci.), vol. 555, pp. 359–370, 1991.

[27] M. Cintra and J. Torrellas, "Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors," in *Proc. 8th Int. Symp. High-Perform. Comput. Archit. (HPCA)*, 2002, pp. 43–54.

[28] M. J. Garzarán et al., "Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors," *ACM Trans. Archit. Code Optim.*, vol. 2, no. 3, pp. 247–279, 2005.

[29] X. Li et al., "Speculative parallel threading architecture and compilation," in *Proc. Int. Conf. Workshops Parallel Process. (ICPP)*, 2005, pp. 285–294.

[30] L. Gao et al., "SEED: A statically greedy and dynamically adaptive approach for speculative loop execution," *IEEE Trans. Comput.*, vol. 62, no. 5, pp. 1004–1016, 2013.

[31] C. Tian et al., "Copy or discard execution model for speculative parallelization on multicores," in *Proc. 41st IEEE/ACM Int. Symp. Microarchit. (MICRO-41)*, Nov. 2008, pp. 330–341.

[32] C. Tian et al., "Enhanced speculative parallelization via incremental recovery," in *Proc. 16th ACM Symp. Principles Practice Parallel Program. (PPoPP)*, 2011, pp. 189–200.

[33] C. Ke et al., "Safe parallel programming using dynamic dependence hints," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Languages Appl. (OOPSLA)*, 2011, pp. 243–258.

[34] H. E. Ramadan et al., "Committing conflicting transactions in an STM," in *Proc. 14th ACM SIGPLAN Symp. Principles Practice Parallel Program. (PPoPP)*, 2009, pp. 163–172.

Álvaro García-Yágüez received the MSc degree in computer science and the MSc degree in research in information and communication technologies from the University of Valladolid, Spain, in 2010 and 2011, respectively. His research interests include parallel and distributed computing and automatic parallelization of sequential code. He is the cofounder of Novagecko (http://www.novagecko.com), a software company in the field of mobile application development.

Diego R. Llanos received the MS and PhD degrees in computer science from the University of Valladolid, Spain, in 1996 and 2000, respectively. He is Associate Professor of Computer Architecture at the Universidad de Valladolid. His research interests include parallel and distributed computing, automatic parallelization of sequential code, and embedded computing. He is a recipient of the Spanish government's national award for academic excellence. He is a Senior Member of the IEEE Computer Society and Member of the ACM.

Arturo Gonzalez-Escribano received the MS and PhD degrees in computer science from the University of Valladolid, Spain, in 1996 and 2003, respectively. He is an Associate Professor of Computer Science at the Universidad de Valladolid. His research interests include parallel and distributed computing, parallel programming models, and embedded computing. He is a Member of the IEEE Computer Society and Member of the ACM.