An OpenMP Extension that Supports Thread-Level Speculation

Sergio Aldea, Alvaro Estebanez, Diego R. Llanos, *Senior Member, IEEE,* and Arturo Gonzalez-Escribano

Abstract—OpenMP directives are the *de-facto* standard for shared-memory parallel programming. However, OpenMP does not guarantee the correctness of the parallel execution of a given loop if runtime data dependences arise. Consequently, many highly-parallel regions cannot be safely parallelized with OpenMP due to the possibility of a dependence violation. In this paper, we propose to augment OpenMP capabilities, by adding Thread-Level Speculation (TLS) support. Our contribution is threefold. First, we have defined a new *speculative* clause for variables inside parallel loops. This clause ensures that all accesses to these variables will be carried out according to sequential semantics. Second, we have created a new, software-based TLS runtime library to ensure correctness in the parallel execution of OpenMP loops that include *speculative* variables. Third, we have developed a new GCC plugin, which seamlessly translates our OpenMP *speculative* clause into calls to our TLS runtime engine. The result is the ATLaS C Compiler framework, which takes advantage of TLS techniques to expand OpenMP functionalities, and guarantees the sequential semantics of any parallelized loop.

Index Terms—Parallelism and concurrency, code generation, thread-level speculation, optimistic parallelization

1 Introduction

The advent of multicore technologies in the new century made parallel processing ubiquitous. Many parallel languages and parallel extensions to sequential languages have been proposed to exploit the capabilities of modern multicore systems. The most successful proposal is OpenMP [1], a directive-based parallel extension to sequential languages (such as C, Fortran or C++) that allows parallel execution of user-defined code regions.

Figure 1 shows an example of (a) a sequential C loop, and (b) its parallelization with OpenMP directives. As can be seen, all variables inside the loop body should be classified as private or shared. Informally speaking, variables whose values are always set in a given iteration before their use should be labeled as private, while variables that have values visible by all threads executing the loop in parallel should be classified as shared. In our example, `a[]` is a read-only shared vector, while `v[]` is a shared vector that is modified by each iteration.

As OpenMP is a simple and powerful mechanism for code parallelization, its use has several limitations. First, the classification of all variables inside the critical region, according to their use, is a time-consuming, error-prone task. Second, OpenMP does not ensure the parallel execution of the code according to sequential semantics, as the programmer is responsible for such a task. In the example shown in Fig. 1, the programmer is responsible for ensuring that each thread modifies a different element of `v[]`. Third, in many cases, potentially-parallel regions cannot be safely parallelized because their control flow depends on runtime data. Consider the code depicted in Fig. 2. Suppose that the value of `k` is not known at compile time. Assuming `b>0` for a given `i`, if the parallel execution of the loop calculates iteration `i` *before* iteration `i-b`, access to `v[i-b]` may return an outdated value, breaking sequential semantics. The only way to guarantee a correct behavior would be to serialize the execution of iterations $i - b$ and i, a difficult task in the general case.

Safely parallelizing loops that may present runtime dependence violations can have a significant impact in terms of performance. We have previously measured the amount of loop-level parallelism that could be extracted from the SPEC CPU 2006 benchmark, with different

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

```
#pragma omp parallel for \
 private (i,b) shared (a,v)
for (i=0; i<MAX; i++) {
   b = func(i);
   v[i] = b * a[i];
}
```
(b)

Fig. 1. Example of loop parallelization with OpenMP.

```
for (i=0; i<MAX; i++) {
   b = func(i);
   if (b==k)
      v[i] = v[i-b];
   else
      v[i] = b * a[i];
}
```
(a)

```
#pragma omp parallel for \
 private (i,b) shared (a,k) \
 speculative(v)
for (i=0; i<MAX; i++) {
   b = func(i);
   if (b==k)
      v[i] = v[i-b];
   else
      v[i] = b * a[i];
}
```
(b)

Fig. 2. A loop that cannot be safely parallelized with current OpenMP clauses (a), and its parallelization with our new speculative clause (b).

• *S. Aldea, A. Estebanez, D. R. Llanos, and A. Gonzalez-Escribano are with Dpto. Informática, Universidad de Valladolid, Campus Miguel Delibes, 47011, Valladolid, Spain.*
E-mails: {sergio,diego,arturo}@infor.uva.es, palestebanez1@gmail.com

techniques [2]. Our results show that, while around 48% of the loops present in the applications analyzed (representing around 13% of their aggregate execution time) are potentially parallelizable with existent parallel programming models such as OpenMP, an additional 38% of loops (representing around 20% of the execution time) could be run in parallel with the help of runtime speculative parallelization techniques.

Our proposal consists in augmenting OpenMP with software-based, Thread-Level Speculation (TLS) techniques to ensure that definitions and uses of shared variables are carried out according to sequential semantics. This solution allows the OpenMP programming model to be used even when dependence violations may arise at runtime. To do so, we define a new speculative clause. Variables labeled as speculative will be accessed following two simple rules:

- All reads of a speculative variable will return the most up-to-date value for this variable. This value can either be generated previously by this thread or by any of its *predecessors*, defined as threads that execute earlier iterations according to sequential semantics. This is called a *forwarding* operation.
- All writes to a speculative variable will store the value in a local copy, and will check whether a *successor* thread (that is, threads that are executing "future" iterations) has consumed an outdated value of this variable. In this case, the offending thread (and possibly some of its successors) will be stopped and re-started, in order to force them to consume the updated value of the variable. This is called a *squash* operation.

As long as a dependence violation forces the values of speculative variables to be discarded, all threads maintain version copies of the speculative variables being accessed. When a *non-speculative* thread (that is, a thread with no alive predecessors) successfully finishes the execution of its block of consecutive iterations, all changes are committed to the main copy of all speculative variables. After this commit operation, the thread will become the *most speculative* one, since it will execute the following block of iterations that remains unassigned.

The three main contributions of this paper are the following:

1) We have defined an extension to OpenMP specifications, adding a clause to support speculative accesses to data in omp parallel for constructs. This clause follows the guidelines proposed by Aldea *et al.* [3].
2) We have created a brand-new TLS runtime library that handles the parallel execution of loops that includes speculative variables, including support for speculative access of pointer-based data of any size without the need for 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.
3) Finally, we have developed a new plugin-based compiler pass to the GCC OpenMP implementation to support the speculative clause. This pass transforms the loop to be parallelized, inserting the runtime TLS calls needed to (a) distribute blocks of iterations among processors, (b) perform speculative loads and stores of speculative variables, and (c) perform partial commits of the correct results calculated so far.

The result is ATLaS, a complete framework that allows OpenMP to execute loops in parallel without the need of a prior dependence analysis. Our performance evaluation, using both synthetic and real-world applications on a real multicore system, shows that this approach leads to performance speedups.

The rest of the paper is organized as follows. Section 2 introduces TLS key concepts. Section 3 describes some related work. Section 4 briefly describes our proposal of a new OpenMP speculative clause. Section 5 describes in detail the architecture of our new TLS runtime library. Section 6 shows how we have added support to handle our new clause in the GCC OpenMP compiler. Section 7 presents the experimental evaluation. Finally, Sect. 8 summarizes our conclusions.

2 Thread-Level Speculation

Speculative parallelization (SP), also called Thread-Level Speculation (TLS) or Optimistic Parallelization [4], 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 given thread generates a datum that has already been consumed by a successor in the original sequential order. In this case, the results calculated so far by the successor (called the offending thread) are not valid and should be discarded. Early proposals [5], [6] stop the parallel execution and restart the loop serially. Other proposals stop the offending thread and all its successors, re-executing them in parallel [7], [8], [9], [10]. A third option (see e.g. [11], [12], [13]) is to only re-start the offending thread and subsequent threads that have actually consumed any value from it, leading to a noticeable performance improvement in some cases.

Figure 3 shows an example of thread-level speculation. The figure represents four threads executing fragments of four consecutive iterations of the same loop. The value of `x` was not known at compile time, so the compiler was not able to ensure that accesses to the `SV` structure do not lead to dependence violations when executing them in parallel. However, the actual values of `x` for each iteration are known at runtime.

Under speculative execution, each thread maintains a version copy of the data structure that is accessed speculatively (here, the `SV` vector). At compile time, the original code is augmented to perform speculative

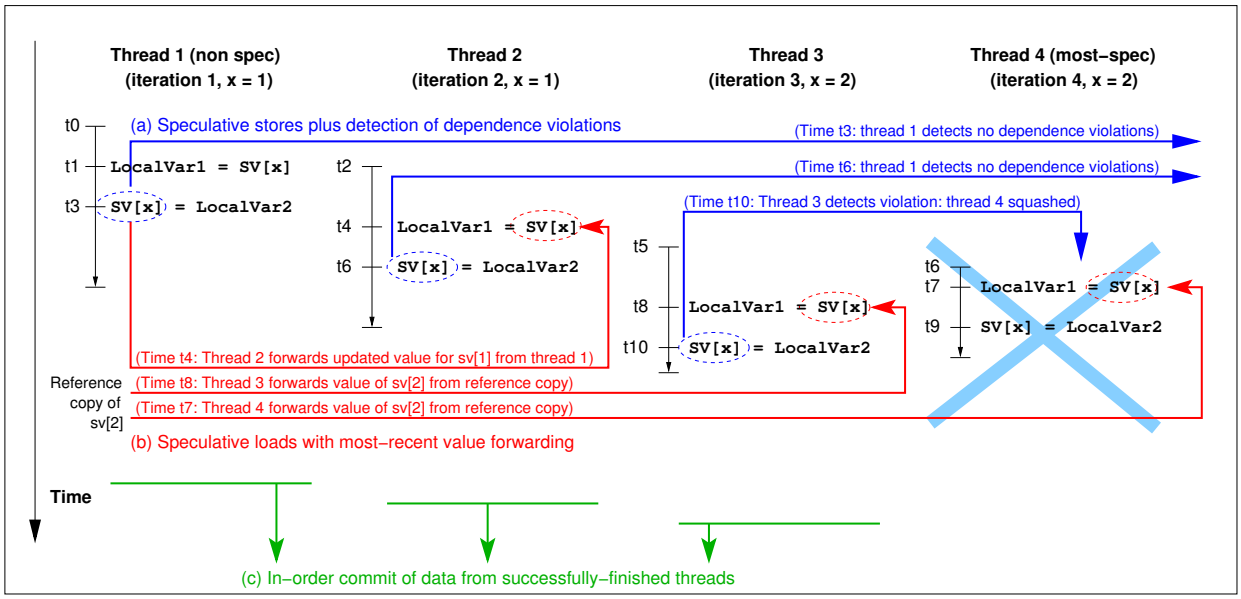

Fig. 3. Example of speculative execution of a loop and summary of operations carried out by a runtime TLS library.

stores, speculative loads, and in-order commits. In addition, the loop structure is rearranged in order to allow the re-execution of squashed iterations. The following paragraphs describe these operations in more detail.

Speculative stores At compile time, all write operations to the data structure being speculatively accessed should be replaced with a *speculative store* function. This function writes the datum in the version copy of the current thread, and ensures that no thread executing a subsequent iteration has already consumed an outdated value for this structure element, a situation called "dependence violation". If such a violation is detected, the offending thread and its successors are stopped and restarted. In the example depicted in Fig. 3, the checks for dependence violations performed by Threads 1 and 2 do not find any successor that has consumed an outdated value for `SV[1]`. However, at time t_{10}, Thread 3 discovers that Thread 4 has already consumed an outdated value for `SV[2]`, so a dependence violation has been found. Therefore, Thread 4 should be stopped and restarted, in a so-called *squash* operation. When Thread 4 is restarted, it will forward the updated value for `SV[2]` from Thread 3, being able to continue the execution of the iteration assigned to it.

Speculative loads At compile time, all reads to the speculative data structure are replaced by a function that performs a *speculative load*. This function obtains the most up-to-date value of the element being accessed. If a predecessor (that is, a thread executing an earlier iteration) has already read or written that element, the value is *forwarded* (as Thread 2 does in Fig. 3). If not, the function obtains the value from the reference copy of the data structure (as Thread 3 does in the figure).

Commit-or-discard operation If no dependence violation arises during the execution of a given thread, its changes to the speculative data structure should be *committed* to the reference copy of the data structure. Note that commits should be done in order, to ensure that the most up-to-date values are stored. In the case of a dependence violation, the intermediate results calculated by this thread should be discarded, an operation known as *thread squash*. In both cases, the scheduling runtime system should assign a new block of iterations to the thread to continue the parallel work.

Scheduling iterations under TLS The scheduling method used with speculative parallelization is different from classic scheduling methods, e.g. [14], [15], [16]. Under TLS, the execution of an iteration or chunk of iterations can be discarded, so the scheduling method should be able to re-assign the squashed iteration to the same or a different thread. The loop structure should be changed to allow re-execution of iterations.

3 Related Work

Software-based TLS (STLS) proposals Several works propose speculative parallelization mechanisms that benefit from different degrees of code transformations. Tian *et al.* [17] 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. 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. In [18], a software-based TLS system is proposed to help in the manual parallelization of applications. The system requires 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 threads 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 are 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 [19], relying on dependence hints provided by the programmer to allow explicit data communication between threads, thus reducing runtime dependence violations. In [9], a model that combines different techniques such as thread-level speculation, helper threads and run-ahead execution is proposed to dynamically choose the most appropriate combination at runtime. A work of the CorD group [20] aims to reduce the cost of misspeculation, by recording 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.

Oancea *et al.* developed *SpLIP* [21], an STLS approach centered on decreasing overheads of speculative operations. In this work, load and store operations directly work with the main copy of the variables, and dependences are managed through exceptions. They extract many of the ideas from software Transactional Memory (STM), implementing non-locking operations where possible, and preserving a log of variables and timestamps to handle the execution. *ATLaS'* runtime library and *SpLIP* are both STLS implementations that can extract speed-up from sequential applications with complex dependences. Conceptually, the main difference between *ATLaS'* runtime library and *SpLIP* is the way they manage their operations, since *ATLaS* manages version copies, while *SpLIP* works with the main version of speculative data. *ATLaS* also incorporates a compile-time phase that greatly simplifies the use of speculation for production purposes. To take advantage of SpLIP, the user has to rewrite the entire application almost from scratch, since the code to be parallelized and the underlying library are extremely highly coupled. ATLaS compile-time and runtime features are mature enough to be used in production environments with almost no effort.

Finally, an adaptive approach for speculative loop execution, which handles nested loops, has recently been proposed [10]. Our proposal does handle nested loops transparently, in the same way standard OpenMP does.

TLS and Software Transactional Memory Both TLS and software Transactional Memory (STM) [22] are solutions that use speculative techniques to improve the programmability and performance of programs. TLS has several features in common with TM, such as the use of speculative reads and writes that can be rolled back. However, and despite their implementation similarities, they solve different problems. The goal of TM is to help in explicit parallel programming by reducing the costs of the locks required to avoid race conditions in critical sections [23], [24]. On the other hand, TLS departs from a sequential program, breaks it into tasks and tries to execute them optimistically in parallel, while preserving sequential semantics.

The main difference between TLS and TM is that TLS ensures a total order in the commit operation, which is always carried out sequentially from the non-speculative to the most-speculative thread. As long as TM does not preserve any order in the commit operations, STM libraries cannot be used directly to mimic the behavior of loop-based speculative parallelization whenever sequential semantics should be preserved. Section 1 of the Supplemental Material further discusses this issue.

Finally, there are several interesting TLS-TM hybrid approaches. These solutions are reviewed in Sect. 2 of the Supplemental Material.

TLS extensions to OpenMP Early works, such as [25], propose the use of OpenMP directives to enable speculative parallelism, the details of the implementation being transparent to the programmer. In a similar way, [26] exposes the advantages of using OpenMP to give explicit hints to the compiler and the underlying hardware to extract speculative parallelism.

Other proposals aim to integrate Transactional Memory technologies into OpenMP (see [27], [28], [29], [30], [31], [32], [33], [34], [35]). These proposals are reviewed in Sect. 3 of the Supplemental Material.

4 Semantics of our *speculative* clause

The problem of adding speculative parallelization support to OpenMP can be handled using two approaches. The first one requires the addition of a new directive, such as `pragma omp speculative for`. However, there are many OpenMP related components that should be modified in order to add a new directive. A simpler solution is to add a new OpenMP clause to the list of available parallel constructs, which allows the programmer to enumerate which variables should be handled speculatively. The syntax of this clause is:

```
speculative(variable[, var_list])
```

In this way, if the programmer is unsure about the use of a certain data structure, he can simply label it as speculative. In this case, a tailored OpenMP implementation should replace all definitions and uses of this data structure with the corresponding `specload()` and `specstore()` function calls. An additional `commit_or_discard()` function will be automatically inserted once each thread has finished its chunk of iterations, to either commit the results, or to restart the execution if the thread has been squashed due to a runtime dependence violation.

Our new TLS runtime library, described in the following section, was indeed developed using standard OpenMP clauses. In order to integrate our library into an experimental OpenMP framework that includes a new speculative clause, two particularities of our TLS library should be taken into account. First, since our TLS runtime library has also been developed using OpenMP, some private and shared control variables should be added to the target loop in order to use it. Therefore, if a speculative clause is found by the compiler, this occurrence, which implies the use of our speculative library, should trigger the inclusion of several private and shared variables to the existing lists. As long as OpenMP allows the repetition of clauses, so the compile time support for this new speculative clause can add additional private and shared clauses that will later be expanded by the compiler.

Second, the standard scheduling methods implemented by OpenMP are not enough to handle speculative parallelization. These methods assume that the execution of a chunk of iterations will never fail, so they do not consider the possibility of restarting a chunk that has failed due to a dependence violation. Therefore, it is necessary to use a speculative scheduling method. Instead of dividing the iteration space, we have followed the solution adopted in [7], replacing the original loop structure with a new loop composed by N iterations, N being the number of threads. At the beginning of the loop, each thread is assigned a different chunk of iterations to be executed. If a thread has successfully finished a chunk, it will receive a new chunk that has not yet been successfully executed. In the case of a dependence violation that triggers a squash operation, the scheduling method will try to reassign to that thread the chunk whose execution has failed, in order to improve locality and cache reutilization.

5 A NEW RUNTIME LIBRARY FOR TLS

We have developed a new TLS runtime library that supports the speculative execution of for loops. The library architecture follows the design principles of the speculative parallelization library developed by Cintra and Llanos [7], [36]. In order to understand our solution, a brief description of that proposal is needed.

In [7], [36], Cintra and Llanos developed a runtime library that uses a sliding window mechanism that allows the parallel execution of W consecutive chunks of iterations. Each time the non-speculative thread finishes, a partial commit takes place; the thread executing the following chunk becomes the new, non-speculative thread; and the window advances, allowing the execution of new chunks of iterations. Despite its good performance figures, the runtime library developed by Cintra and Llanos suffers from severe limitations. First, their library requires all speculative variables to be packed in a single, one-dimensional vector before the start of the speculative loop. Second, all speculative variables should share a single data type. Third, speculative variables can only be accessed by name inside the loop (no references by addresses or pointers were allowed). Finally, this runtime library creates W version copies of the entire speculative data structure, being W the size of the sliding window being used, instead of just keeping version copies of the data elements actually accessed. These limitations prevent the use of this runtime library to support a speculative clause, where variables and data structures labeled as speculative may be of different data types, can be accessed by name or address, and where speculative data structures can be of any size.

Our TLS runtime library overcomes all these limitations. It allows variables of any data type to be speculatively accessed, both by name or address, and managing the space needed for version copies on demand. In this section, we will briefly show the general architecture of the library. A more detailed description of the design decisions faced can be found in [37].

5.1 Loop transformation for speculative execution

Figure 4 briefly shows the transformation of a parallel loop for speculative execution. This transformation is triggered by our proposed speculative clause, and it is automatically carried out by our compiler plugin. The changes are briefly described below:

- **Line 1:** Additional, internal variables are defined.
- **Line 2:** Before the loop, the omp_set_num_threads() function is called to define the number of threads to be used.
- **Line 3:** A specbegin() function is called to initialize the execution of the following parallel loop. If it is the first loop being parallelized, this function also initializes the runtime speculative library.
- **Line 4:** All variables labeled as speculative are automatically reclassified as shared. Besides this change, all reads and stores inside the *loop body* on those speculative variables (see below) are replaced with calls to specload() and specstore() functions, in order to keep sequential consistency, as described in Sect. 2. Our compiler plugin also labels other internal variables needed by the runtime systems as private and shared, 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, thus reaching the end of the thread (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. 4(a) is replaced with a call to the specload()

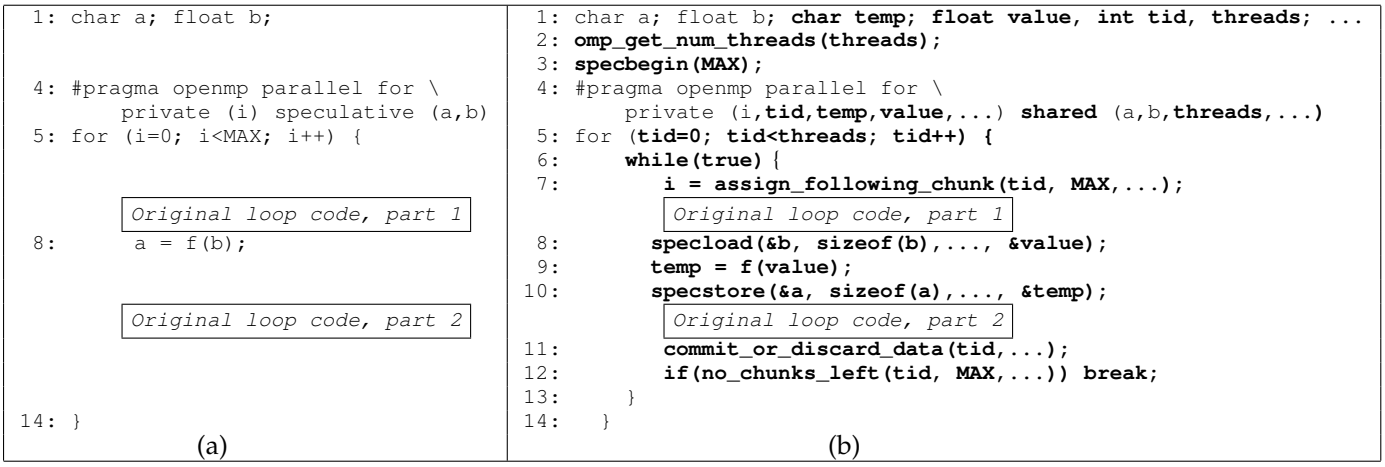

Fig. 4. Loop transformation to allow its speculative execution: Original (a) and transformed (b) code.

function, which 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. 4(a) also performs a write on a. This write is replaced with a call to specstore() (line 9), which 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 [13]) are squashed.
It is important to highlight that only the lines of the original loop body that involve speculative variables are changed in this way: the remaining code is left with no changes.

- **Line 11:** Once the original loop body is finished, 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. 5.4.
- **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.

5.2 Data structures

The data structures needed by the new speculative library are depicted in Fig. 5(a). The sliding window mechanism is implemented by a matrix with W window slots (four in the figure). Each slot acts as a "scratchpad" used to handle the speculative execution of a particular chunk of iterations. Two global variables, non-spec and most-spec, indicates the slot assigned to the execution of the non-speculative and most-speculative chunks of iterations at each particular moment. These variables are used as limits to stop the search for predecessor versions and the search for possible dependence violations, respectively. 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 will be executed in parallel using three threads. It is very important to understand that there is no fixed association between threads and slots. Whenever a thread is assigned a new chunk of iteration, it is also assigned the corresponding slot to work in. This allows an order relationship to be maintained between the chunks being executed.

In our example, the thread working in slot 1 is executing the non-speculative chunk of iterations (as indicated by its RUNNING state); the following chunk has already been executed and its data has been left there to be committed after the non-spec chunk finishes (since it is in the 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 cannot 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. 5.4, 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 5(a) represents a situation where the programmer declared three variables within

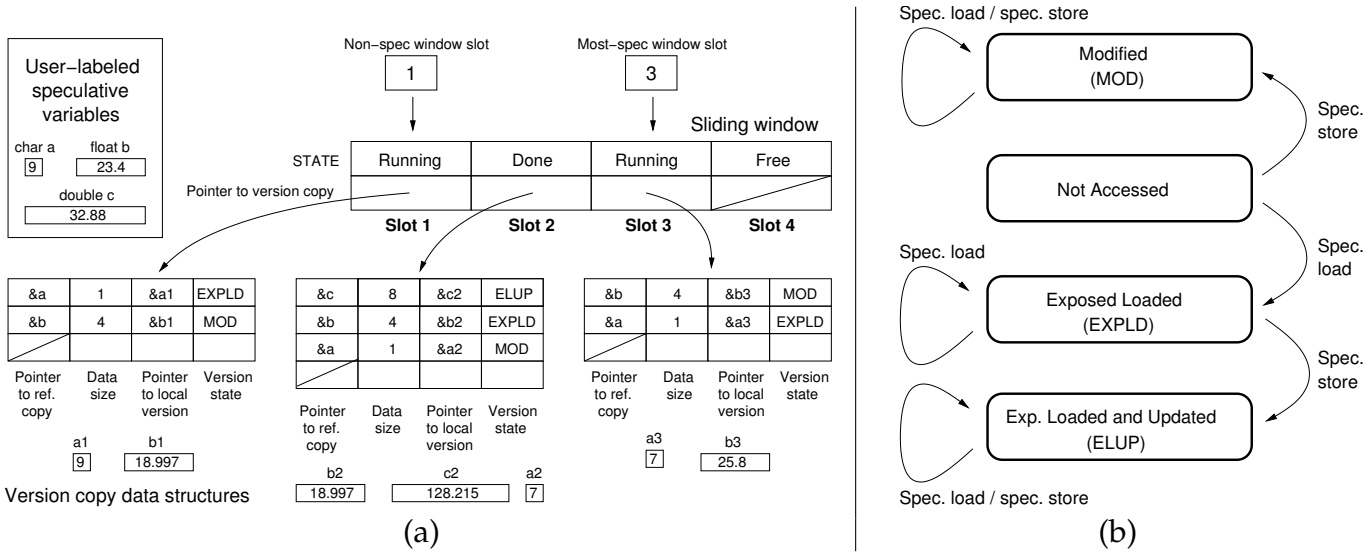

Fig. 5. Data structures of our new speculative library (a) and state transition diagram for speculative data (b).

our speculative clause. 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. Note that, although entire data structures may be labeled as speculative, speculative reads and writes are always carried out over scalar variables. Therefore, the maximum size of the data being speculatively accessed will be the size of the biggest scalar variable in the architecture considered. This value is 8 bytes in 64-bit architectures. The third column 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. 5(b).

Figure 5(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 the thread working in slots 3 has forwarded its value. With respect to variable b, the information in the figure shows that b was overwritten by both threads working in slots 1 and 3.

5.3 Speculative loads and stores

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

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

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

Recall that specload() should return the most up-to-date value available for the speculative variable. Figure 6 shows how the speculative load works. Suppose that the thread working in slot 2 has only accessed to variable c so far, and it then calls specload(&b, sizeof(b), 2, &value) to obtain a value for b. The sequence of events is the following:

1) The thread working in slot 2 scans its version copy data structure to check whether a value for b has been already stored there. As long as the only speculative variable accessed so far is c, this search produces no results.
2) Our thread goes to its predecessor version copy data structure and scans it in order to find a value for b. Its predecessor has stored a value for it, so our thread copies its value to a new location. Note that, if no value for b were found there, our thread would go to the next predecessor, until the non-speculative thread is found. If no predecessor had used the value, our thread would get the value from the reference copy.
3) After storing a copy of b's value, the thread working in slot 2 adds a new row to its version copy data structure, storing the address of b, its data size, the address of the version copy of b being managed by the thread, and the new state for this version copy, EXPLD. The call to specload() finishes by returning

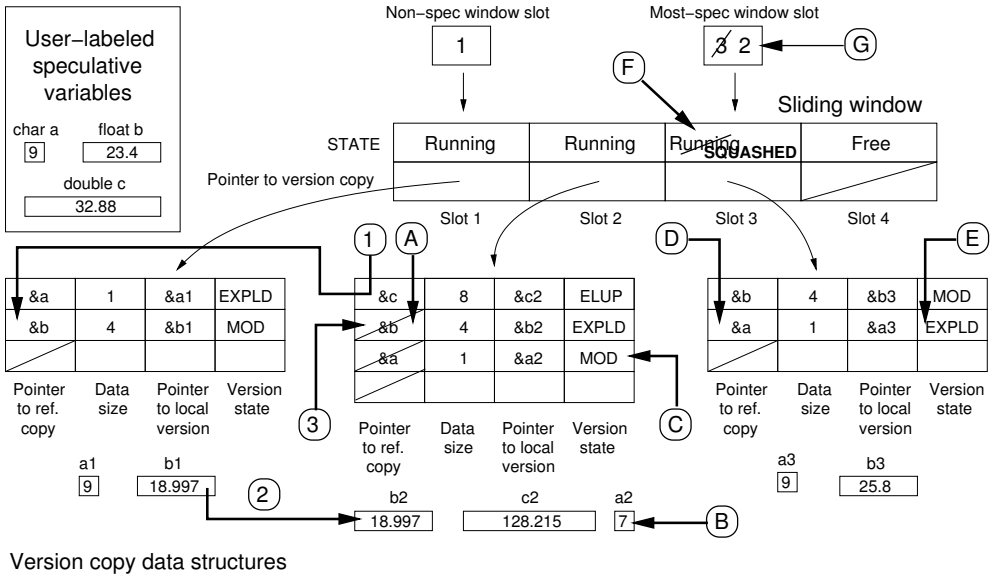

Fig. 6. Steps of a speculative load (1..3) and speculative store (A..F).

the value 18.997 in the address indicated by its fourth parameter.

The interface of specstore() is the same as specload(), 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.

Figure 6 shows the sequence of events related to a speculative store. Suppose that the thread working in slot 2 executes specstore(&a, sizeof(a), 2, &temp), where temp holds the value 7. The sequence of events is the following:

A) The thread working in slot 2 searches for a local version copy of a. At this moment, only copies of c and b are stored in its version copy data structure, so the search produces no results. If a were found, this thread would update its status according to the state diagram of Fig. 5(b), and it would proceed to step D.
B) The thread working in slot 2 creates a local copy of a, storing value 7 on it.
C) A new row is added to the version copy data structure, with a pointer to a, its size, the pointer to the local copy and the status, which, in this case, will be MOD (see Fig. 5(b)).
D) After storing the value locally, the thread working in slot 2 should check whether any successor has consumed an outdated value. To do so, our thread would scan (in increasing order of speculativeness) for any successor slot that holds a copy of a in the EXPLD or ELUP states. These states would indicate that the successor has used the value. In our example, the search finds out that the thread working in Slot 3 has consumed an incorrect value for a. If no dependence violation was detected, the call to specstore() would finish here.
E) A dependence violation has been detected. Thread working in slot 3 should be squashed. To do so, the thread working in slot 2 changes the state of slot 3 from Running to Squashed. Since all threads check their own state at the beginning of each specload(), specstore(), and at the end of the execution of each chunk of iterations, thread working in slot 3 will eventually discover that it has been squashed, and will execute a call to commit_or_discard() to be assigned a new chunk (possibly the same) and start the process again.
F) Finally, the thread working in slot 2 marks itself as the most-speculative thread, since data stored in association with slot 3 is no longer valid. The most-spec pointer will be advanced later by the thread that receives the task of re-executing chunk 3.

If, after these events, the thread working in slot 2 finishes its execution, while the threads associated to slots 1 and 3 are still working, we reach the situation shown in Fig. 5(a). Note that, at that point, the thread working in slot 3 has already been re-started and it has forwarded the most up-to-date value for a (that is, 7) from slot 2.

5.4 Partial commit operation

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

Suppose that we are in the situation depicted in Fig. 5(a), and the non-spec thread working in slot 1 finishes. As long as it is the non-spec one, it will scan its data structure for variables in the ELUP or MOD states. In our example, b has been modified, so it copies the

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

It is interesting to note that each thread only writes on its local version copy data structure, so no critical sections are needed to protect them. The only critical section used protects the sliding window data structure, to avoid that a thread overwrites another thread's state.

5.5 Performance hurdles

One of the main advantages of our new speculative parallelization library is that each thread only allocates the memory needed to store local copies of the speculative data *actually* being accessed (see step (3) of the speculative load operation and step (B) of the speculative store, above). In contrast, Cintra *et al.*'s solution keeps T copies of the entire list of speculative variables. As will be seen in Sect. 7.2, the number of potentially-speculative variables can be huge, so Cintra *et al.*'s solution severely limits scalability.

Our improvement in terms of memory footprint 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. T being the number of threads, in [7], the time complexity of this operation was in $T \times O(1) = O(T)$, since all the memory needed for any data that might be accessed was allocated in advance. In our scheme, N being 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 are somewhat lower than the ones described in [7].

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, defined by the programmer (see [38] for more details). Before accessing the data, a module 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. We are also evaluating other solutions, such as dichotomic search, which can be used to reach search values in $O(T.\log(N))$, but it comes at the cost of spending more time finding the place to store the data locally.

6 Compiler support for the *speculative* clause

The compiler phase of our system is implemented on the GCC C compiler [39], extending its functionality through a plugin. Before describing the implementation of the plugin, it is necessary to introduce the GCC architecture.

GCC architecture in a nutshell Figure 7 shows the scheme of the GCC architecture [40], [41]. In basic terms, GCC is a big pipeline that converts one program representation into another, in different stages. Each stage generates a lower-level representation, until the assembly code is generated at the last stage. GCC architecture has three clearly-defined blocks: *Front End*, *Middle End* and *Back End*. There is one front end for each programming language. The parser of each language converts source files into a unified tree form, called GENERIC, which is a high-level tree representation. When it finishes, the Front End emits a GENERIC intermediate representation (IR) of the code, which serves as the interface between the front end and the rest of the compiler.

The Middle End works on GIMPLE, which is a 3-address language with no high-level control flow structures. In GIMPLE, each statement does not contain more than three operands (except function calls); control flow structures are combinations of conditional statements and goto operators; and there is a single scope for variables. This kind of representation is convenient to optimize the source code. Once the source code is in GIMPLE form, an *interprocedural optimizer* is called, where *inlining* operations, *constant propagation*, or *static variable analysis* are performed. We have inserted our plugin at this point.

The following step is the transformation from GIMPLE into SSA (Static Single Assignment) representation. In SSA form, each variable is assigned or written only once, creating new versions for each assignment of the same variable, which can be read many times. When different versions of the same variable are written into both branches of a conditional expression, a ϕ-function is added just after the conditional block, allowing the selection of the correct version of the variable, depending on the branch executed. SSA representation is used for several optimizations, such as forward expression substitution, loop interchange, vectorization or parallelization, among others. These optimizations are performed in around 100 passes.

After these optimizations, the SSA representation is converted back to the GIMPLE form, which is transformed into a *register-transfer language* (RTL) form, in which the Back End works on. RTL was the original primary intermediate representation used by GCC. It is a

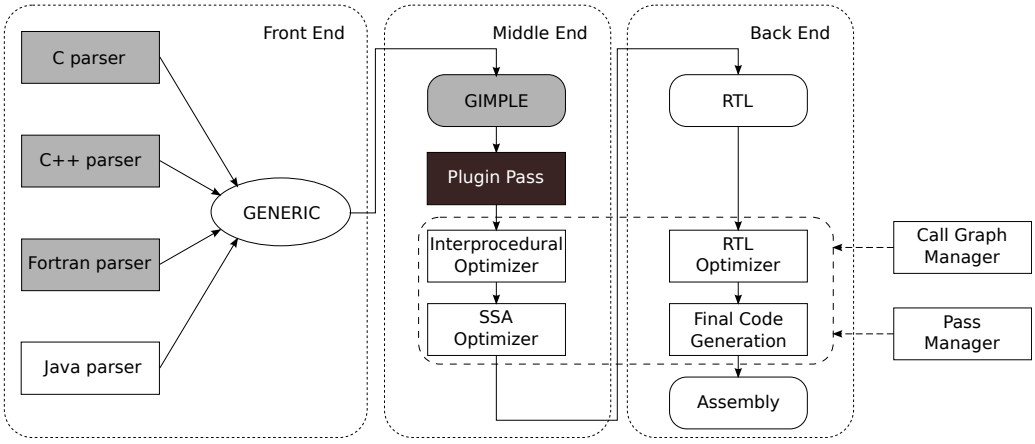

Fig. 7. GCC Compiler Architecture. The main OpenMP related components, highlighted in grey, are the C, C++ and Fortran parsers, and the GIMPLE IR level. Highlighted in black is the location of our plugin pass.

hardware-based representation which corresponds to an abstract machine with an infinite number of registers. GCC also uses this form to perform several optimizations, such as branch prediction or register renaming, in around 70 passes.

Finally, the *Final Code Generation* step of the Back End creates the assembly code for the target architecture (x86, mips, etc.) from the RTL representation.

Transactions between the different phases are sequenced by the *Call Graph* and the *Pass Manager*. The Call Graph Manager generates a call graph for the compilation unit, decides in which order the functions are optimized, and drives the interprocedural analysis. The Pass Manager sequences individual transformations and handles pre- and post-cleanup actions as needed by each pass.

Parsing the new clause In order to parse the new speculative clause, we have extended the GNU OpenMP (GOMP) compiler, the OpenMP implementation for GCC. The main parts of the GCC architecture related with OpenMP are highlighted in grey in Figure 7. GOMP has four main components [30]: parser; intermediate representation; code generation; and the runtime library called `libGOMP`. We have focused on modifying the GOMP parsing phase. The generation of new code to support TLS is located in the plugin developed, and mainly consists of inserting calls to the TLS library functions described in the previous sections.

The parser identifies OpenMP directives and clauses, and emits the corresponding GENERIC representation. We have modified the C parser and the IR to add support for the new speculative clause. First, we have created the GENERIC representation of the new clause as other standard clauses. Then, the compiler has been modified to recognize and parse that clause as part of the parallel loop construct. When the new clause has been parsed, and the IR is generated, our plugin detects the clause and triggers all the transformations needed by the code.

GCC speculative plugin description GCC plugins provide extra features to the compiler –although they cannot extend the parsed language–, allowing passes to be added, replaced, monitored, or even removed from the GCC compiler without touching the GCC source code. Hence, plugins ease the programming of modifications and contributions to the GCC community. Using this mechanism, our system adds a new pass in the GCC pipeline. This new pass performs all the transformations needed in the code when the programmer marks a variable as speculative.

The new pass is added before the compiler optimization passes, and just before GCC does the first pass in relation with OpenMP: *omplower*. At this point, we have the code in a GIMPLE representation, and the for loop marked with the parallel loop directive preserves all the clauses introduced by the programmer. Therefore, we have the information about which variables are speculative. After this pass, GCC manages speculative variables as shared, while their handling as speculative is carried out by the TLS runtime library.

Figure 4 shows a brief example of the transformations made by the plugin. The parser detects the new speculative clause, and the new compiler pass performs automatically all the transformations needed to speculatively parallelize the loop. With the list of variables and data structures that should be speculatively updated, the plugin replaces each read of one of these variables or data elements with a specload() function call. Similarly, all write operations to speculative variables are replaced with a specstore() function call. Loads or stores involving other variables do not require additional changes in the code, since all flavors of private and shared variables keep their respective semantics in the context of a speculative execution. The plugin also adds all the structures and functions needed to speculatively parallelize the code. This process is completely transparent to the programmer, who does not need to know anything about the speculative parallelization model. The programmer should only label the variables involved in the target loop as private or shared, as with any other OpenMP

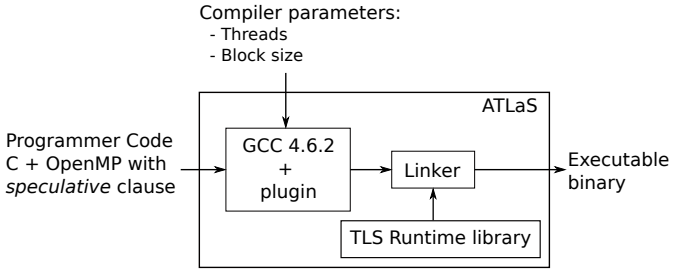

Fig. 8. Overview of the code generation process for the speculative clause

program, and mark as speculative those variables that might lead to any dependence violation.

The scheme of the process followed by the plugin can be summarized in the following steps:

1) The plugin traverses each function of the original program looking for an OpenMP parallel loop directive with a speculative clause on it. If the plugin does not find the speculative clause on the pragma, the semantic of the loop remains identical to any other standard OpenMP loop.
2) If the plugin finds the speculative clause, it extracts the speculative variables pointed to by the clause, and two functions are added before the loop: omp_set_num_threads(T), where T is the number of threads indicated in the compilation command; and specbegin(N), where N is the number of iterations of the loop.
3) The plugin adds, as private or shared variables, those variables needed by the runtime system. The code generated by the plugin also includes the creation of other new variables, which are also added as private or shared.
4) The plugin adds all the code needed to run the TLS system, including the replacement of the original loop by a new loop that drives the speculative execution.
5) The plugin traverses the GIMPLE nodes of the loop, searching for readings from and writings into the speculative variables. Each read and write are replaced by a specload() and specstore() function, respectively.

Once the plugin has transformed the loop, GCC operation continues with the next passes. When the compilation ends, the resulting binary file is prepared to run speculatively.

Use of the ATLaS framework To speculatively parallelize a source code with our system, programmers should add the OpenMP directive in the target loop, and classify its variables, according to their usage, into private (and its variants), shared, or speculative. To compile the program, the programmer should also indicate the size of the block of iterations that will be issued for speculative execution, among other minor parameters. With these simple modifications, a programmer can speculatively parallelize a code, while the rest of the transformations needed are transparently performed by the plugin and the compiler. Figure 8 summarizes the code generation process performed by the plugin, and the link to the TLS runtime system, which is transparent to the user.

7 Experimental evaluation

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

7.1 Real-world benchmark evaluation

To test the ATLaS framework, we have used both real-world and synthetic benchmarks. The real-world applications include the 2-dimensional Minimum Enclosing Circle (2D-MEC) problem, the 2-dimensional Convex Hull problem (2D-Hull), the Delaunay Triangulation problem, and a C implementation of the TREE benchmark. The synthetic benchmarks are described in the Supplemental Material.

The 2D-MEC problem consists in finding the smallest circle that encloses a set of points. We have parallelized the randomized incremental approach due to Welzl [42], which 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 its frontiers. It is interesting to note that points inside the old solution may lie outside the new one. Therefore, all points should be processed again to check if the new circle encloses them. The solution can be defined by two or three points, and the algorithm is composed of three nested loops. We have used a random, ten-million point, uniformly distributed input set. We have speculatively parallelized the innermost loop, which consumes 43.75% of the total execution time (see Tab. 1). The 2D-Hull problem solves the computation of the convex hull (smallest enclosing polygon) of a set of points in the plane. We have parallelized Clarkson *et al.* [43]'s implementation. The algorithm starts with the triangle composed by the first three points and adds points in an incremental way. If the point lies inside the current solution, it will be discarded. Otherwise, the new convex hull is computed. Note that any change to the solution found so far generates a dependence violation, because other successor threads may have used the old enclosing polygon to process the points assigned to them. The probability of a dependence violation in the 2D-Hull algorithm depends on the shape of the input set. Therefore, we have used three different, ten-million-point input sets to run this benchmark. The

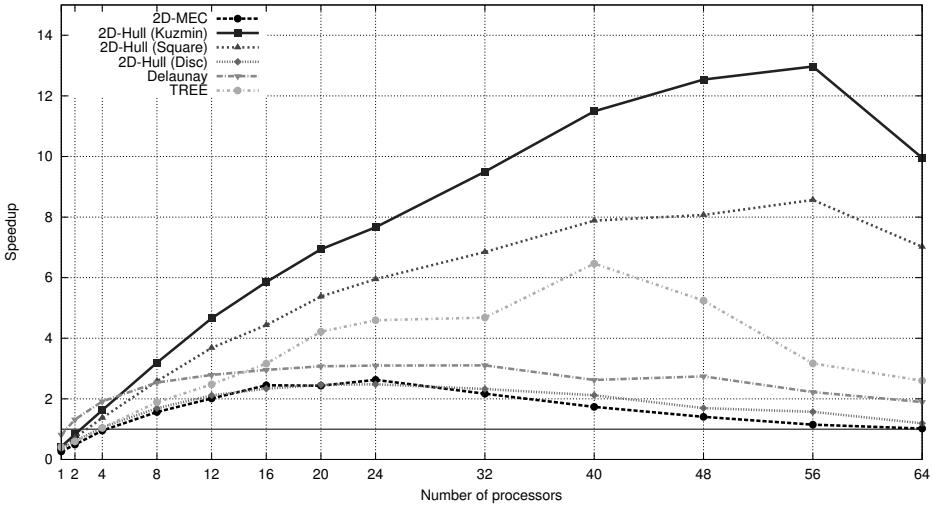

Fig. 9. Performance achieved by the parallelizable loop of the benchmarks considered.

Kuzmin input set follows a Gauss-Kuzmin distribution, with a higher density of points around the center of the distribution space, which leads to very few dependence violations, since points far from the center are very scarce. The two other input sets, *Square* and *Disc*, cause more dependence violations than *Kuzmin*, with their points uniformly distributed inside a square and a disc, respectively. The *Square* input set leads to an enclosing polygon with fewer edges than the *Disc* input set, thus generating fewer dependence violations.

The next real-world application is the randomized incremental construction of the Delaunay Triangulation using the Jump-and-Walk strategy, which was introduced by Mücke et al. [44], [45]. This incremental strategy starts with a number of points, called anchors, whose containing triangles are known. The algorithm finds the closest anchor to the point to be inserted (the jump phase), and then traverses the current triangulation until the triangle that contains the point to be inserted is found (the walk phase). The goal of the algorithm is to find the network of triangles in which all the circumcircles of all triangles in the network are empty, i.e., the circumcircle of each triangle contains no other vertices than those three that define the triangle. We have used an input set of 5000 anchors, and one million points to be inserted.

The TREE problem [46], unlike the previous three applications, does not suffer from dependence violations, but it is still not parallelizable at compile time because the compiler is not able to ensure that there are no data dependencies. Compilers also find hurdles in several sum and maximum reductions contained in the code, which ATLaS detects and handles properly. We have run this benchmark with a 4096-point input set.

Figure 9 shows the speedups achieved using the proposed OpenMP speculative clause with the mentioned real-world applications. For the 2D-MEC benchmark, our solution achieves a peak speedup of $2.6\times$. Although these are not big figures, these results are achieved by simply declaring as speculative the variables that hold the solution found so far.

In the case of 2D-Hull, as described above, results depend on the input set. Performance varies from a $2.4\times$ speedup with the *Disc* input set, which causes a huge number of dependence violations, to a $13\times$ speedup with the *Kuzmin* input set, which leads to fewer violations.

Delaunay's execution produces a high number of dependence violations, which affects the speedup. Delaunay achieves a peak performance of $3.1\times$ speedup.

Finally, TREE obtains a peak of $6.5\times$ speedup. This benchmark is characterized by the presence of reductions over sum and maximum operations that involve speculative variables.

Performance comparison with other TLS solutions A recent paper of our group [37] helps to put these results into perspective. That work compares the performance of the ATLaS runtime library with respect to the TLS library developed by Cintra and Llanos [7]. The results described in [37] show that the current version of the ATLaS runtime system achieves 68% to 75% of the speedup obtained by Cintra and Llanos' library. Recall that, as we described in Sect. 5, the applicability of Cintra and Llanos' solution is severely limited, while ATLaS is of general use. Regarding *SpLIP* [21], as mentioned in Sect. 3, its use implies a rebuilding of the entire application to tightly integrate their approach into the sequential code, a work that exceeds the objectives of this paper.

7.2 Effectiveness of the ATLaS runtime library

Table 1 summarizes the percentage of time consumed by the target loops of each benchmark, together with an estimation of the maximum speedup obtained (using Amhdahl's Law), and the performance results obtained by our runtime library for the entire application, both in terms of speedup and as a percentage of the maximum speedup attainable. The last two columns indicate the

Application	% Target loop	Max. speedup with P=64 (Amhdahl)	Maximum speedup obtained	% of exploited speedup	% of iterations that present dep. violations	# of potentially speculative scalar variables	Size of chunks issued
2D-MEC	43.75	1.76	1.37	**77.84%**	0.009%	10	1 800
FAST	100	64	44.49	**69.52%**	0.001%	2	25
TREE	95.17	15.84	5.12	**32.32%**	0%	259	100
2D-Hull, Kuzmin	100	64	12.92	**20.18%**	0.0008%	1 206	11 000
2D-Hull, Square	100	64	8.47	**13.23%**	0.0032%	3 906	3 000
Delaunay	97.60	25.47	2.96	**11.62%**	0.5%	12 030 060	2
2D-Hull, Disc	100	64	2.48	**3.88%**	0.0219%	26 406	1 250

TABLE 1
Percentages of parallelism effectively exploited by ATLaS for the benchmarks considered, together with some benchmarks' characteristics. I/O time consumed by the benchmarks were not taken into account.

percentage of iterations that lead to runtime dependence violations, and the number of speculative variables. Since all benchmarks but TREE present dependences among some iterations, the value given by Amhdahl's law is just an upper bound of the available parallelism.

The percentage of the speedup effectively exploited depends on a number of factors. The first one is the occurrence of runtime dependence violations. In general, the more dependences there are, the less speedup there will be. This fact can be observed in the results for the execution of 2D-Hull with different input sets that lead to a different number of runtime dependencies. The second factor is load imbalance, since not all iterations present the same amount of workload. As long as the scheduling mechanism implemented in ATLaS issues chunks of iterations of fixed size (with the best sizes obtained by experimentation), runtime load imbalance is not being mitigated in any way. The third factor that affects parallel performance is the efficiency of the ATLaS runtime library itself. The performance results obtained with the FAST benchmark, described in the Supplemental Material, show that the library presents a very low runtime overhead. Finally, the fourth factor is the number of speculative variables. As can be seen, the more speculative variables there are, the less percentage of exploited speedup there will be. This is due to the cost of the commit operation, which should be done sequentially for each variable by the non-speculative thread.

A more detailed analysis of the TLS operations carried out by the library, together with an execution breakdown for speculative loads and stores, can be found in [37]. Regarding the influence of the number of squashes in performance, please see [13]. The ATLaS framework incorporates tools to measure these and other values. Please refer to the ATLaS documentation for more details.

8 Conclusions

The ATLaS framework allows loops that cannot be analyzed at compile time and/or can present dependence violations when executed in parallel to be easily parallelized. The solution consists in the use of a new speculative clause to point out the variables that may lead to dependence violations. The use of this solution does not require more knowledge than the use of standard OpenMP directives. Moreover, its use simplifies the task of classifying variables according to their usage: If a programmer is unsure about the feasibility of the parallel execution of a given loop, he/she may label as speculative the variables that affect it. Such a decision guarantees the correct parallel execution of the loop, possibly at the cost of a lower performance. Our solution not only executes loops in parallel correctly, even when runtime dependence violations arise, but also achieves noticeable speedups in applications not parallelizable by other means.

Our future work also includes offering different scheduling mechanisms in addition to the fixed-size chunking currently implemented, and other squashing alternatives in addition to inclusive squash.

The ATLaS framework is freely available at **http://atlas.infor.uva.es**.

Acknowledgments

We would like to thank the anonymous reviewers and the Editor for their work. This research is partly supported by the Castilla-Leon Regional Government (VA172A12-2, PIRTU); 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] R. Chandra *et al.*, *Parallel Programming in OpenMP*, 1st ed. Morgan Kaufmann, Oct. 2000.
[2] S. Aldea *et al.*, "The Bonafide C analyzer: Automatic loop-level characterization and coverage measurement," *The Journal of Supercomputing*, vol. 68, no. 3, pp. 1378–1401, june 2014.
[3] ——, "Support for thread-level speculation into OpenMP," in *IWOMP'12 Proceedings*, June 2012, pp. 275–278.
[4] M. Kulkarni *et al.*, "Optimistic parallelism requires abstractions," in *PLDI'07 Proceedings*, 2007, pp. 211–222.
[5] M. Gupta and R. Nim, "Techniques for speculative run-time parallelization of loops," in *SC'98 Proceedings*, 1998, pp. 1–12.
[6] L. Rauchwerger and D. Padua, "The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization," in *PLDI'95 Proceedings*, 1995, pp. 218–232.

[7] M. Cintra and D. R. Llanos, "Toward efficient and robust software speculative parallelization on multiprocessors," in *PPoPP'03 Proceedings*, June 2003, pp. 13–24.

[8] F. H. Dang *et al.*, "The r-lrpd test: Speculative parallelization of partially parallel loops," in *IPDPS'02 Proceedings*, 2002, pp. 20–29.

[9] P. Xekalakis *et al.*, "Combining thread level speculation helper threads and runahead execution," in *ICS'09 Proceedings*, 2009, pp. 410–420.

[10] L. Gao *et al.*, "Seed: A statically-greedy and dynamically-adaptive approach for speculative loop execution," *IEEE Transactions on Computers*, vol. 62, no. 5, pp. 1004–1016, May 2013.

[11] X.-F. Li *et al.*, "Speculative parallel threading architecture and compilation," in *ICPPW '05 Proceedings*. IEEE Computer Society, 2005, pp. 285–294.

[12] C. Tian *et al.*, "Speculative parallelization using state separation and multiple value prediction," in *ISMM '10 Proceedings*. New York, NY, USA: ACM, 2010, pp. 63–72.

[13] A. García-Yágüez *et al.*, "Squashing alternatives for software-based speculative parallelization," *IEEE Transaction on Computers*, vol. 63, no. 7, pp. 1826–1839, july 2014.

[14] T. Hagerup, "Allocating independent tasks to parallel processors: An experimental study." *J. Parallel Distrib. Comput.*, vol. 47, no. 2, pp. 185–197, 1997.

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

[16] T. H. Tzen and L. M. Ni, "Trapezoid self-scheduling: A pratical scheduling scheme for parallel compilers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 1, pp. 87–98, 1993.

[17] C. Tian *et al.*, "Copy or discard execution model for speculative parallelization on multicores," in *MICRO-41 Proceedings*, nov. 2008, pp. 330–341.

[18] C. Ding *et al.*, "Software behavior oriented parallelization," in *PLDI'07 Proceedings*, 2007, pp. 223–234.

[19] C. Ke *et al.*, "Safe parallel programming using dynamic dependence hints," in *OOPSLA'11 Proceedings*, 2011, pp. 243–258.

[20] C. Tian *et al.*, "Enhanced speculative parallelization via incremental recovery," in *PPoPP'11 Proceedings*, 2011, pp. 189–200.

[21] C. E. Oancea *et al.*, "A lightweight in-place implementation for software thread-level speculation," in *SPAA '09 Proceedings*. ACM, 2009, pp. 223–232.

[22] N. Shavit and D. Touitou, "Software transactional memory," *Distributed Computing*, vol. 10, pp. 99–116, 1997.

[23] L. Ceze *et al.*, "Bulk disambiguation of speculative threads in multiprocessors," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 227–238, 2006.

[24] J. a. Barreto *et al.*, "Unifying thread-level speculation and transactional memory," in *Proceedings of the 13th International Middleware Conference*, ser. Middleware '12. New York, NY, USA: Springer-Verlag New York, Inc., 2012, pp. 187–207.

[25] J. F. Martínez and J. Torrellas, "Speculative synchronization: applying thread-level speculation to explicitly parallel applications," in *ASPLOS'02 Proceedings*, vol. 37, Oct. 2002, pp. 18–29.

[26] V. Packirisamy and H. Barathvajasankar, "Openmp in multicore architectures," *University of Minnesota, Tech. Rep*, 2005.

[27] W. Baek *et al.*, "The OpenTM transactional application programming interface," in *ISCA'07 Proceedings*, 2007, pp. 376–387.

[28] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *ISCA'93 Proceedings*, 1993, pp. 289–300.

[29] J. Larus and C. Kozyrakis, "Transactional memory," *Communications of the ACM*, vol. 51, no. 7, pp. 80–88, Jul. 2008.

[30] D. Novillo, "OpenMP and automatic parallelization in GCC," in *Proceedings of the 2006 GCC Developers' Summit*, Ottawa, Canada, 2006, pp. 135–144.

[31] M. Milovanović *et al.*, "Transactional memory and OpenMP," in *IWOMP'07 Proceedings*, 2007, pp. 37–53.

[32] ——, "Multithreaded software transactional memory and OpenMP," in *MEDEA'07 Workshop*, 2007, pp. 81–88.

[33] C. Ferri *et al.*, "SoC-TM: integrated HW/SW support for transactional memory programming on embedded MPSoCs," in *CODES+ISSS'11 Proceedings*, 2011, pp. 39–48.

[34] M. Wong *et al.*, "A case for including transactions in OpenMP," in *IWOMP'10 Proceedings*, 2010, pp. 149–160.

[35] IBM, "Thread-level speculative execution for C/C++," 2012, tech. report.

[36] M. Cintra and D. R. Llanos, "Design space exploration of a software speculative parallelization scheme," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 6, pp. 562–576, 2005.

[37] A. Estebanez *et al.*, "New data structures to handle speculative parallelization at runtime," ser. HLPP 2014 Proceedings, July 2014.

[38] ——, "Improving the perfomance of a pointer-based, speculative parallelization scheme," in *PPGM'14 Proceedings*, February 2014.

[39] GNU Project, "GCC, the GNU Compiler Collection," http://gcc.gnu.org/, [Last visit: March 2014].

[40] ——, "GCC internals," http://gcc.gnu.org/onlinedocs/gccint/, [Last visit: March 2014].

[41] D. Novillo, "GCC an architectural overview, current status, and future directions," in *Proceedings of the Linux Symposium*, Tokyo, Japan, September 2006, pp. 185–200.

[42] E. Welzl, "Smallest enclosing disks (balls and ellipsoids)," in *New results and new trends in computer science*, ser. Lecture notes in computer science, vol. 555. Springer-Verlag, 1991, pp. 359–370.

[43] K. L. Clarkson *et al.*, "Four results on randomized incremental constructions," *Comput. Geom. Theory Appl.*, vol. 3, no. 4, pp. 185–212, 1993.

[44] E. P. Mücke *et al.*, "Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations," in *SoCG'96 Proceedings*, 1996, pp. 274–283.

[45] L. Devroye *et al.*, "A note on point location in Delaunay triangulations of random points," *Algorithmica*, vol. 22, pp. 477–482, 1998.

[46] J. E. Barnes, "TREE," Jan. 1997, institute for Astronomy, University of Hawaii. http://www.ifa.hawaii.edu/~barnes/ftp/treecode/.

Sergio Aldea received his M.Sc. in Computer Science and his M.Sc. in Research in Information and Communication Technologies from the Universidad de Valladolid, Spain, in 2010, and 2011, respectively. His research interests include parallel and distributed computing, automatic parallelization of sequential code, and automatic code generation. More information about his current research activities can be found at http://www.infor.uva.es/∼sergio.

Alvaro Estebanez received his M.Sc. in Computer Science and his M.Sc. in Research in Information and Communication Technologies from the Universidad de Valladolid, Spain, in 2012, and 2013, respectively. His research interests include parallel and distributed computing, automatic parallelization of sequential code.

Diego R. Llanos received his MS and PhD degrees in Computer Science from the University of Valladolid, Spain, in 1996 and 2000, respectively. He is a recipient of the Spanish government's national award for academic excellence. Dr. Llanos is Associate Professor of Computer Architecture at the Universidad de Valladolid, and his research interests include parallel and distributed computing, automatic parallelization of sequential code, and embedded computing. He is a Senior Member of the IEEE and Senior Member of the ACM. More information about his current research activities can be found at http://www.infor.uva.es/∼diego.

Arturo Gonzalez-Escribano received his MS and PhD degrees in Computer Science from the Universidad de Valladolid, Spain, in 1996 and 2003, respectively. Dr. Gonzalez-Escribano is Associate Professor of Computer Science at the Universidad de Valladolid, and 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. More information about his current research activities can be found at http://www.infor.uva.es/∼arturo.