
UNIVERSIDAD DE VALLADOLID

ESCUELA TÉCNICA SUPERIOR

INGENIEROS DE TELECOMUNICACIÓN

TRABAJO FIN DE MÁSTER

MÁSTER UNIVERSITARIO EN INVESTIGACIÓN

EN TECNOLOGÍAS DE LA INFORMACIÓN Y LAS COMUNICACIONES

# Improving the Perfomance of a Pointer-Based, Speculative Parallelization Scheme

Autor:

**D. Álvaro Estébanez López**

Tutores:

**Dr. D. Diego R. Llanos Ferraris**
**Dr. D. Arturo González Escribano**

Valladolid, 19 de Julio de 2013

| | |
|---|---|
| TÍTULO: | **Improving the Perfomance of a Pointer-Based, Speculative Parallelization Scheme** |
| AUTOR: | **D. Álvaro Estébanez López** |
| TUTORES: | **Dr. D. Diego R. Llanos Ferraris**<br>**Dr. D. Arturo González Escribano** |
| DEPARTAMENTO: | **Informática (ATC, CCIA, LSI)** |

**Tribunal**

| | |
|---|---|
| PRESIDENTE: | **Dr. D. Jesus María Vegas Hernández** |
| VOCAL: | **Dr. D. Joaquín Adiego Rodríguez** |
| SECRETARIO: | **Dr. D. David Escudero Mancebo** |
| FECHA: | **19 de Julio de 2013** |
| CALIFICACIÓN: | |


## Resumen del TFM

La paralelización especulativa es una técnica que intenta extraer paralelismo de los bucles no paralelizables en tiempo de compilación. La idea subyacente es ejecutar el código de forma optimista mientras un subsistema comprueba que no se viole la semántica secuencial. Han sido muchos los trabajos realizados en este campo, sin embargo, no conocemos ninguno que fuese capaz de paralelizar aplicaciones que utilizasen aritmética de punteros. En un trabajo previo del autor de esta memoria, se desarrolló una librería software capaz de soportar este tipo de aplicaciones. No obstante, el software desarrollado sufría de una limitación muy importante: el tiempo de ejecución de las versiones paralelas era mayor que el de las versiones secuenciales. A lo largo de este Trabajo de Fin de Máster, se aborda esta limitación, encontrando y corrigiendo las razones de esta falta de eficiencia, y situando el trabajo realizado en perspectiva, dentro de las contribuciones mundiales en este ámbito.

Los resultados experimentales obtenidos con aplicaciones reales nos permiten afirmar que estas limitaciones han sido solventadas, ya que obtenemos speedups de hasta de un $1.61\times$. Así, con la nueva versión de la librería se han llegado a obtener mejoras de hasta el 421.4% respecto al tiempo de ejecución generado por la versión original de la librería especulativa.



## Palabras clave

Paralelización especulativa, Paralelismo.

## Abstract

Speculative parallelization is a technique that tries to extract parallelism of those that can not be parallelized at compile time. The underlying idea is to optimistically execute the code in parallel, while a subsystem checks that sequential semantics have not been violated. There exists many researchs in this field, however, to the best of our knowledge, there are not any solution that allows to effectively parallelize those applications that use pointer arithmetic. In a previous work, the author of this Master Thesis developed a software library that allows the parallelization of this kind of applications. Nevertheless, the software developed has an important limitation: Execution time of the parallelized versions was higher than the sequential one. Along this Master Thesis, this limitation is addressed, finding and solving the reasons of this lack of efficiency and putting the author's own work in perspective with respect to international contributions in the field.

Experimental results obtained with real applications allow us to affirm that this limitation has been overcome since we have achieve speedups of $1.61\times$. In this way, the new library version has improved the execution time with respect to original version of the library in a 421.4%.

## Keywords

Thread-level Speculation, Parallelism.

# Agradecimientos

A lo largo de la realización de mi Trabajo de Fin de Máster he recibido una gran apoyo de las personas de mi alrededor. Con este pequeño texto me gustaría agradecerles toda la ayuda recibida.

Primero, en el ámbito más académico, me gustaría dar las gracias a los miembros del grupo Trasgo de la UVa. Me gustaría hacer especial mención de Diego R. Llanos para agradecerle todas las recomendaciones que me ha dado, sus consejos me han servido tanto para el desarrollo de este Trabajo de Fin de Máster, como en el ámbito personal, gracias a él he podido mejorar intentando superarme cada día.

Pero por otro lado, en el desarrollo de un proyecto más o menos duradero, no todo son ayudas sobre temas estrictamente académicos, es necesario también otro tipo de apoyo. En este aspecto, primero me gustaría recordar a Pilar López, mi madre y la mejor persona que conozco. A ella le debo todo lo soy, donde he llegado, y donde llegaré. Gracias por enseñarme que con empeño y trabajo duro se pueden sacar las cosas adelante. Tampoco quisiera olvidarme de Juan Cruz Estébanez, mi hermano, que sin querer tantas cosas me ha enseñado. Asimismo quiero agradecer a mi novia, Tania Alonso, los ratos que me ha animado, aguantado, y aconsejado. Además no podría dejar de mencionar a mis abuelos Marcelo López y Priscila Alonso, por toda su ayuda a lo largo de mi vida. Gracias por lo que me habéis dado, siempre os estaré agradecido. Me gustaría agradecer también a los demás familiares y amigos que aún sin una cita textual, también han estado a mi lado.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

*A lo largo de la historia de los procesadores siempre se ha deseado obtener velocidades mayores para obtener mejores tiempos de respuesta. Sin embargo, en la actualidad existen límites que no pueden ser sobrepasados, y por tanto, tenemos que recurrir a que varios procesadores se repartan tarea para actuar como si fuesen uno más rápido, es decir, que trabajen de forma paralela. Este trabajo trata sobre la paralelización especulativa, un tipo de paralelismo que asume, de forma optimista, que todos los bucles son paralelizables, es decir, que cada iteración de un bucle se puede realizar concurrentemente. Por tanto, este tipo de paralelización se centra en los casos que no se pueden paralelizar en tiempo de compilación porque presentan dependencias entre los datos. Así, esta técnica reparte las iteraciones entre los procesadores disponibles, y monitoriza que no se produzcan violaciones de dependencia entre los datos. Si esto ocurre descarta los resultados fraudulentos y reinicia dicha ejecución. Podemos considerar que aparece una violación de dependencia cuando se obtiene un dato diferente del que se obtendría en una ejecución secuencial. Por tanto, para llevar a cabo este tipo de ejecución necesitamos un mecanismo hardware o software que vigile las dependencias entre iteraciones.*

*Este tipo de paralelización necesita una serie de modificaciones en el código original de los programas que permitan (a) distribuir bloques de iteraciones entre los procesadores, (b) obtener o escribir los valores adecuados de las variables que presenten dependencias a través de las correspondientes funciones, y (c) consolidar los resultados al final de las ejecuciones, es decir, que la versión final de las variables tenga los datos que la ejecución secuencial impondría.*

*A lo largo del capítulo, además de introducir la paralelización especulativa, se proporciona la descripción de varios trabajos previos realizados por el grupo de investigación entre los que cabe destacar "Desarrollo de un motor de paralelización especulativa con soporte para aritmética de punteros", un trabajo realizado como Trabajo de Find de Grado, donde se elaboró un esquema basado en software que posibilita la paralelización especulativa extendiendo la funcionalidad de los motores existentes. Sin embargo, ésta librería de software especulativo, tenía más funcionalidad, pero sus tiempos de ejecución no eran buenos, es decir, no se producía ganancia de las versiones*

*paralelas, respecto a las secuenciales.*

*Por tanto, este Trabajo de Fin de Máster (TFM) pretende responder a la pregunta de investigación: "¿Es posible mejorar la versión actual del motor de paralelización especulativa para obtener mejoras del rendimiento en alguna de las aplicaciones disponibles?"*

*Hecha la pregunta, cabe mencionar los objetivos del TFM entre los que podemos destacar: (a) Mejorar el rendimiento actual del motor especulativo para conseguir resultados mejores que los obtenidos para las versiones secuenciales, y (b) Extender la base de aplicaciones en las que probar dicho motor.*

*Por último, describiremos la estructura que seguirá este informe: el capítulo 2 describe el Estado del Arte en el campo de la paralelización especulativa. Los capítulos 3 y 4 son una extensión del Estado del Arte, ya que contienen una descripción más exhaustiva de los principales trabajos de investigación sobre los que se basa este TFM. El capítulo 5 muestra cómo se localizaron los cuellos de botella existentes en la arquitectura de paralelización especulativa existente, así como las mejoras aplicadas para solventarlos. El capítulo 6 contiene una descripción de los resultados experimentales, así como de los benchmarks utilizados. Por último, el capítulo 7 concluye este TFM e introduce alguna de las posibles vías de continuación del mismo.*

During processor history, higher speeds have been desired in order to get better response times. However, there exists some limits that can not be exceeded, and then we have to distribute tasks along several processors that act as if they were one.

## 1.1 Motivation

Since the first microchip (called 4004[1]) was issued by Intel™ in 1971 computing machines have suffered several modifications. Nowadays, speed of the processors is achieving very high levels, however, exists an implicit need to increase it. Due to technology advances, we are beginning to reach barriers that can not be overcome, such as speed of light. Chips are becoming more and more sophisticated. Moreover, they are full of transistors and heat dissipation is becoming a big problem: If heat levels are too high, processors may not work properly.

In this context computers with more than one processor have emerged. There are some advantages with the use of multiprocessor computers, such as the possibility of decreasing heat centralization in a single point by separating the calculations through processors. In this way heat may have distinct points of generation and will not be condensed in a single point. Other advantage is related to the limits imposed by speed of light, that could be mitigated if several chips work concurrently to achieve the same issue. Those reasons lead to multiprocessor machines.

On the other hand, we need programs whose instructions may be executed at the same time to achieve high speed ups, i.e., programs whose instructions do not have to be executed sequentially. To determine this point es tedious because we need to take into account many factors to avoid synchronization errors. Nowadays there exist some specific languages to develop this tasks, and also extensions to sequentially languages and function libraries. But some knowledge about underlying hardware and about the problem to be parallelized is needed, in addition to have the mentioned libraries installed. In addition, it will not be useful to develop software to specific architectures because it will not be portable to other machines. So, the best way to parallelize a source code is allowing compilers to do this task. At the present time, there exists some compilers that may parallelize a code, nevertheless its parallelization is not the best possible because current compilers use a conservative mode to parallelize some algorithm. The main reason of this fact, is the existence of dependence violations within program data. An algorithm should have independent instructions to allow its concurrent execution, nevertheless, to deliver a prediction is not that simple. So, in fact, if compilers have the slightest suspicion of the existence of a dependence violation, they do not create parallel codes. Figure 1.1 shows a loop without dependence violations: All the instructions are independent, so, compiler may directly order its parallel execution. On the other hand, Figure 1.2 contains a loop that may produce some dependence violations. Suppose that the value of `k` is not known at compile time. Assuming `k=5`, if the parallel execution of the loop calculates iteration `i(i=5)` before iteration `i-2(i-2=3)`, access to `v[i-2] (v[i-2]=v[3])` may return an outdated value, breaking sequential semantics. The only way to guarantee a correct behavior would be to serialize the execution of iterations $i - 2(i - 2 = 3)$ and $i(i = 5)$, a

[1]The microchip 4004 had about 2300 transistors and a frequency of 104 KHz

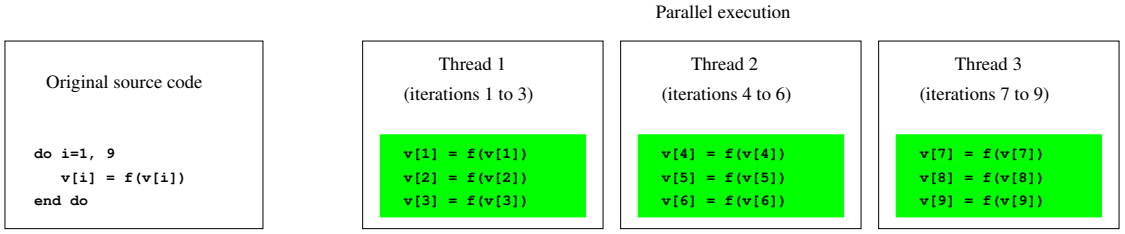



Figure 1.1: Loop without dependences within data of its iterations.

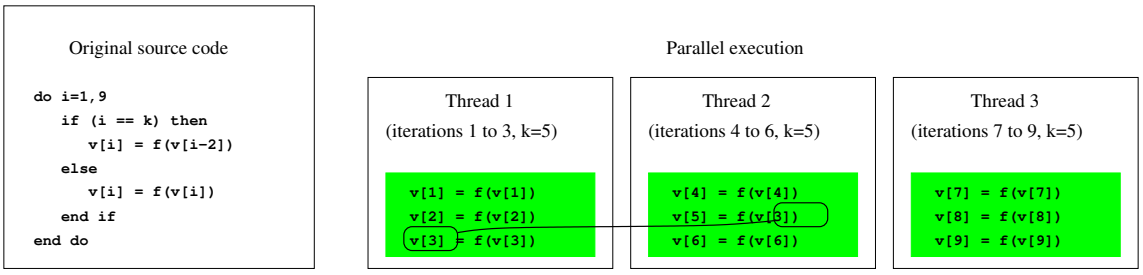



Figure 1.2: Loop with dependences within data of its iterations.

difficult task in the general case. Note that if the dependence did not cross thread boundaries (for example, with `k = 6`), the compiler could parallelize the loop.

Reader at this time may believe that there are some codes that can not be parallelized at compile time, and, in fact, is not misguided at all. However, in spite of the fact that compiler can not create parallel code in many cases, exists some tools that allow this task. Some of those methods are *inspector-executor* and *speculative parallelization*.

### Inspector-executor

This technique [49] allows to parallelize loops that could not be parallelized by a compiler. It is based on the existence of an inspector loop extracted from the original loop whose goal is to find data dependences among data from the iterations of the loop. The set of iterations that depends on each other are assigned to the same processor, in order to ensure its execution in the correct sequence. Hence, loops that may be executed in parallel will be produced.

The use of this method is advised if the processing time of the inspector loop is quite lesser than the execution time of the original loop. However, we should include an exhaustive analysis of data dependences that, in many cases, will be impossible, because it may need input data, or it uses pointer variables, etc. For these reasons, this technique has not be very studied, and can be considered overcome. So, we are going to center our efforts in the other method: *speculative parallelization*.

### Speculative parallelization

Speculative parallelization [28, 29, 34], also called Thread-Level Speculation [10, 11, 16, 52, 57, 69, 70] or Optimistic Parallelization [48], aims to automatically extract loop- and task-level parallelism when a compile-time dependence analysis can not guarantee

```
1 for (i=0; i<100; i++)
2 {
3   localVar = sharedVarA+sharedVarB;
4   sharedVarA = i*localVar;
5 }
```

Figure 1.3: Language C example that shows private and shared variables. localVar is a private variable, in order that is first written, to be read later. However, sharedVarA and sharedVarB are shared variables. In this way, sharedVarB is only read and do not produce dependence violations. On the other hand, shareVarA is read and written, and will produce some dependence violations.

that a given sequential code is safely parallelizable. Speculative parallelization procedure assumes that sequential code can be optimistically executed in parallel, and relies on a runtime monitor to ensure that no dependence violations are produced. A dependence violation appears when a given thread produces 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, and then, this thread is restarted with the correct values. Obviously, some time is lost stopping and re-starting threads, so, the less dependence violations, the better results are obtained, and of course, the more parallel is the loop, the better the results produced with this technique.

The monitoring system may be 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. [13, 39]). On the other hand, systems based on software requires to change the original source code of the loop including some instructions that manage the execution and control possibly dependence violations that take place. Despite the fact of these instructions involve a performance overhead, software-based speculative parallelization can be implemented in current shared-memory systems without any hardware changes.

On the other hand, we need to distinguish between shared and private variables to locate those variables that may cause a dependence violation. An example is shown at Figure 1.3.

Private variables are those whose values are modified at each iteration. Those variables are intended to be used only in the same iteration, that is, the values of those variables are assigned, and read, in the same iteration, and are not used beyond them.

On the other hand, shared variables contain values used in some different iterations during execution time. Therefore, if all variables of the loop are private, we can say that loop is parallelizable. However, if all of them are shared variables, it does not mean that dependence violations will always arise. This situation may appear if a value is modified in a determined iteration and is required in one of the following iterations. At this context, a wrong value of the variable may be obtained because of the lack of security with the execution in the correct order, i.e., as the sequential order pattern. If a thread has to read the value of a shared variable, it must read the most recently value of it. In the case of needing to write over a shared variable, it should be considered whether any of the

following threads have used this value, and in that case, it can be said that a dependence violation arise because a wrong value has been obtained.

There are three types of dependence violations:

1. `Write-after-write (WAW)`: This kind of dependence violation is found when a variable, whose value has been modified in a previous iteration, is written. At this context, we can not ensure that the correct value will be obtained at the end of the execution. Let us see an example written in language `C` to have a better understanding of this situation.

```
for (i=0;i<4;i++)
{
  if (i==1)
    localVar = 4;
  if (i==3)
    localVar = 7;
}
```

   In the example described, the `localVar` variable is written twice, once in iteration `2`, and another one in iteration `4`. In a sequential code, value obtained at the end of the loop in `localVar` would be `7`; however, if we suppose that are available two processors, we do not know whether iteration `2` is executed before iteration `4`, and, therefore can not be ensured that the value of `localVar` at the end of the loop is `7` (the value would be erroneously `4`).

2. `Write-after-read (WAR)`: This kind of dependence violation is found when a local variable that has been read previously, is going to be written with a new value. Therefore, can not be ensure that the correct value of the variable has been read. Let us see an example again.

```
// Suppose that at the beginning
// localVarA == 3
for (i=0;i<4;i++)
{
  if (i==1)
    localVarB = localVarA;
  if (i==3)
    localVarA = 5;
}
```

   There is a read of the variable `localVarA` in iteration `2`. It can also be observed that the same variable is written in iteration `4`. In a sequential code, the value of `localVarA` at the end of the loop would be `5`, and the value of `localVarB` would be `3`. But if we suppose that are two processors available, iterations `1` and `2` are assigned to the first one, and iterations `3` and `4` are assigned to the other processor. We can not know if iteration `2` is executed before iteration `4`, and then we can not ensure that the value of `localVarB` at the end of the loop is `3`. This situation

happens because iteration 4 could be executed before iteration 2, and the value of the variable localVarB would be 5 at the end.

3. Read-after-write (RAW) : This is the most dangerous kind of dependence violation. This error occurs whenever a shared variable is written with a value that would be read by any of the successor threads. When this situation happens, the value read by any of the threads will be wrong. Let us explain this case with the help of an example written in language C:

```
1  // Suppose that at the beginning...
2  // sv[1] = 0;
3  // LocalVar = 1;
4  for (i=0;i<3;i++)
5  {
6    localVar = sv[1];
7    sv[1] = 20;
8    if (i==1)
9    {
10     sv[1] = 10;
11   }
12 }
```

To describe an error situation, suppose that there are three processors, and each one manages a thread. Each thread is going to execute an iteration, namely, thread 1, iteration 0; thread 2, iteration 1; and thread 3, iteration 2. If we execute that code sequentially, values obtained would be localVar = 10 and sv[1] = 20. Nevertheless, with the parallel execution, those values would not be necessarily obtained. Suppose that at instant t1 the first instruction of threads 1 and 2 is executed, so localVar version of each thread should be 0. After that, at instant t2 the following instruction of thread 1 is executed, then sv[1] = 20. Once completed, at instant t3 the first instruction of thread 3 is executed , and the error takes place: localVar version of thread 3, the final version in a sequential execution, would be 20 instead of 10. Moreover, if we suppose that at instant t4 the next instruction of the thread 3 is executed, the value of sv[1] would be 20. And if, at that instant t5 the last instruction of thread 2 is executed, sv[1] would be 10. At the end of the execution, the values of variables would be localVar = 20 or localVar = 0, and sv[1] = 10. All those operations are resumed in the Table 1.1.

Dependence violations described are not irresolvable, to correct them is needed an exhaustive control of the accesses to shared variables, in order to if one of the threads uses a wrong value, will discard it. To get this target, each thread works with its own shared data version, so, each thread modifies its own version of the shared variable, not the global shared variable. When a thread ends the chunk of iterations that have just been executed, it may, or may not, have appeared some dependence violations.

If there were not any errors (no dependence violations appear), results would be saved in the global shared variable. This operation is known as *commit*. On the other hand, if data managed by any of the threads were wrong, all the wrong threads must discard those

| Instant | Thread 1 | | Thread 2 | | Thread 3 | |
|---|---|---|---|---|---|---|
| | $localVar$ | $sv[1]$ | $localVar$ | $sv[1]$ | $localVar$ | $sv[1]$ |
| t1 | 0 | 0 | 0 | 0 | 1 | 0 |
| t2 | 0 | 20 | 0 | 20 | 1 | 20 |
| t3 | 0 | 20 | 0 | 20 | 20 | 20 |
| t4 | 0 | 20 | 0 | 20 | 20 | 20 |
| t5 | 0 | 10 | 0 | 10 | 20 | 10 |

Table 1.1: Timing of the loop with the values of the variable at each instant of time.

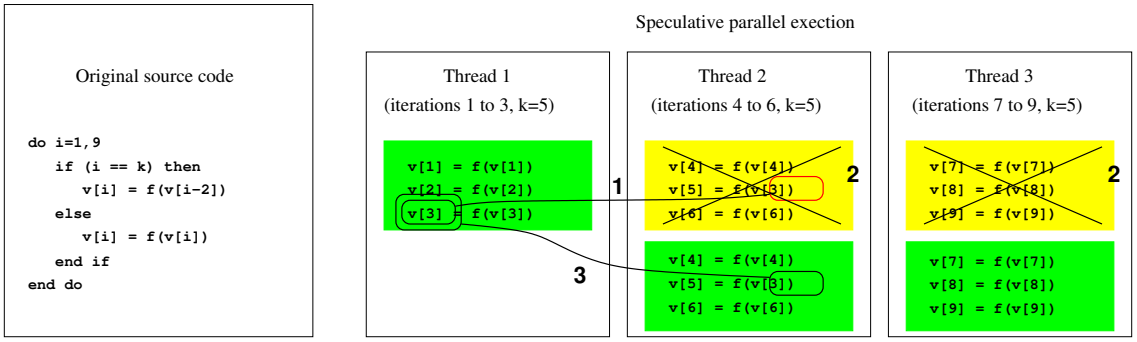



Figure 1.4: Speculative parallelization starts the parallel execution of the loop, while a control system tracks the execution to detect cross-thread dependence violations. If such a violation occurs, (1) speculative parallelization stops the consumer thread and all threads that execute subsequent blocks, (2) discards its partial results, and (3) restarts the threads to consume the correct values.

actual values and start those executions again. The situation described is called a *squash operation*. If any thread is squashed, all the following threads must be discarded too. This process is needed because when a thread read a value do not always use its own local value, but it may consult the previous threads copies, and then, it may consume a wrong value from a squashed thread. An example of this situations can be found at Figure 1.4. Of course, this stop-and-restart process spends some time, so, the more dependence violations appear, the worst results will be obtained with this technique.

### 1.1.1 Modifications in original source code of programs

Software speculative parallelization systems should modify the original source code of the original application at compile time. Some functions are needed to achieve this goal:

- *Distribution of chunks of consecutive iterations:* Iterations should be distributed along all available threads that take part over speculative execution. It can be done with different strategies: Distribute iteration chunks with a constant size; adapting them to the characteristics of each application; or dynamically deciding the size of the next chunk to launch.

- *Speculative load and store operations:* Each thread has its own version of the shared variables, so all read and write operations on shared variables should be replaced

with library functions that also should check that no dependence violations appears.

- *Results commitment:* At the end of a successful execution of a chunk of iterations, a function should be called to commit the produced results, and to request a new chunk of consecutive iterations.

### 1.1.2 Summary

Speculative parallelization is an execution time technique that lets to extract parallelism from sequential codes that can not be analyzed at compile time. The results obtained up to date shows that it can accelerate some sequential codes. However, its scope is reduced by some unsolved questions. These limitations and the fact that automatic parallelism can not take advantage of all the knowledge of programmers about inherently parallel solutions, provoke that nowadays to design and implement parallel programs natively is indispensable to take advantage of the characteristics of parallel systems.

## 1.2 Previous works of the research group

One of the main research interests of the Trasgo group is related to speculative parallelization. The group has already developed a runtime library (known as "engine") that implements the functions needed to handle speculative parallelization. In this context, let us see some of the works developed by the Trasgo group:

- *Toward Efficient and Robust Software Speculative Parallelization on Multiprocessors* [10]. This article contains information about how speculative parallelization technique helps applications to obtain better performance. We can see a software description that implements this technique. The main characteristics of the proposed scheme are a sliding window mechanism, a synchronization policy that relaxes the requirements of critical sections, and data structures that support the executions. With applications that do not have too much dependence violations, the authors show that speedups of 71% against sequential versions can be achieved. Along the document the fundamentals of speculative parallelization are described, what dependence violations are, and what operations should implement a speculative parallelization software. At the end, they execute several applications with some dependence violations, to check the new software efficiency, with a complete study of the times produced by each operation.

- *Design Space Exploration of a Software Speculative Parallelization Scheme* [11]. This document is an extension of the article described above. In this work, the authors test some applications that generate more dependence violations than the applications seen at [10].

- *New Scheduling Strategies for Randomized Incremental Algorithms in the Context of Speculative Parallelization* [56]. This article focuses on how iterations should be scheduled to improve performance of speculative parallelization schemas. It focuses on the incremental randomized algorithms, a kind of applications that is not

easy to parallelize. It introduces the reader to the development of MESETA [55], a scheduling mechanism that takes into account the probability of occurrence of a dependence violation to determine the chunk of iterations to assign.

- *Ejecución paralela de algoritmos incrementales aleatorizados* [35]. This article explores randomized incremental algorithms in more detail. These algorithms have a clear benefit in runtime in the context of Computational Geometry. To achieve a demonstration of this hypothesis, the paper gives two examples: The calculation of the *convex hull*, and the *minimum enclosing circle (MEC)* problem. Tests performed with these applications shows that exists a significant acceleration of parallel runtime against sequential versions.

- *Paralelización especulativa de un algoritmo para el menor círculo contenedor* [25]. This work shows how speculative parallelization was used to speed up the MEC problem. It contains a review of the main variables of the software used to perform this parallelization, and some experimental results.

- *Desarrollo de un motor de paralelización especulativa con soporte para aritmética de punteros* [23]. This paper explains the basis of the implementation that attempts to solve the limitations of the software seen in [10, 11]. We can find a survey of some possible implementations to solve the problem. After that, discuss why they have chosen one of the implementations, instead of the other solutions, and offer some implementation details. Finally, they show some experimental results. However, all of these results are slower than the corresponding sequential execution. To mitigate this issue is the main goal of our work.

## 1.3 Research question

The main problem related to the current version of the speculative library is its lack of performance. Hence the main purpose of this master thesis is to improve its performance:

> ***Is it possible to improve current version of the speculative library in order to obtain speedups in any of the available applications?***

It is mandatory to try to improve the performance of the last version developed, because at this time the speculative versions of the applications are slower than sequential versions, and these results imply that we have a powerful tool without any practice application.

## 1.4 Objectives

The main objectives of this Master thesis are the following:

- The main problem to solve is the lack of performance of the current engine. Actually, speculative parallelization of the applications with the engine does not produce benefits, i.e., the execution time of parallel versions is worse than time of sequential

versions, so, at the moment this engine es not very useful. Therefore at this project we will research how this library could be improved.

- Another part of the work is to use the speculative library with more applications, to have more experimental results.

## 1.5 Document structure

This work is structured as follows: Chapter 2 describes the related work in the area of speculative parallelization. In this way, we analyze proposals and then extract conclusions in order to classify them how could they be automatically parallelized or not. Also, a depth description of the solution purposed by Cintra and Llanos is provided, since it is the work that leads to the new speculative library described in Chapter 3.

Chapter 4 introduces the new speculative library developed in previous works. The main structures that compose this new architecture are described. Furthermore, some details of how the schema works are provided, i.e., descriptions of the main operations that implements this schema.

Chapter 5 shows how we have located the main bottlenecks of the speculative library, and also describes three optimizations developed in order to improve the performance of this speculative architecture.

Chapter 6 contains the experimental evaluation carried out. Specifically, it contains a description of the benchmarks used to develop the experimental results. The experimental results obtained are also shown. At the end of this chapter, a brief evaluation of the results is given.

Chapter 7 resumes this Master thesis, explores the conclusions achieved after its realization, presents some future work, and cite the publications generated so far by this research topic.

Finally, Appendix A describes the content of the CD-ROM attached to this document.

# Chapter 2

# State of the Art

*La descripción del Estado del Arte es uno de los puntos más relevantes de un TFM. Este capítulo resume algunas de las soluciones existentes para llevar a cabo la paralelización especulativa de aplicaciones. Para ello comenzamos hablando de los artículo donde primero se comenzó a hablar de esta técnica de paralelización, entre los que cabe destacar [16, 39, 57, 69, 70, 71].*

*Más tarde introduciremos algunos de los sistemas que implementan esta técnica a través de software, [10, 11, 23, 24, 26, 27, 46, 68, 73, 74, 75, 76, 82].*

*La siguiente sección muestra algunas de las soluciones basadas en modificaciones en el hardware subyacente de las máquinas: [6, 13, 37, 38, 61, 72]*

*Existen técnicas de paralelismo que se basan en la combinación de varias existentes. Así destacamos [33, 40, 47, 58, 66, 77] como artículos que combinan la paralelización especulativa con otras técnicas para lograr mejores tiempos de ejecución.*

*En [45] encontramos una clasificación de los tipos de paralelización especulativa, según las técnicas que se empleen. Los mismos autores proporcionan una descripción del comportamiento de la paralelización especulativa con uno de los benchmarks más utilizados: SPEC CPU2006 [44].*

*Por otro lado, en [11, 28, 30, 83] encontramos algunas de las decisiones de diseño a la hora de elaborar una librería con soporte para especulación. El trabajo [29] introduce cómo podrían tratarse las excepciones en aplicaciones especulativas*

*También cabe mencionar algunas de las soluciones actuales para paralelizar aplicaciones, que van desde librerías con directivas [63], hasta compiladores automáticos o semiautomáticos que paralelizan un código secuencial especulativamente [4, 5, 80], así como sistemas que detectan los bucles que podrían beneficiarse de la paralelización especulativa [19, 20, 21].*

*Finalmente, en uno de los trabajos más recientes mencionados, se resume una solución que aplica las técnicas de paralelización especulativa al contexto de las GPUs [85].*

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

## 2.1 Origins of the speculative parallelization

Runtime speculative parallelization in software was introduced in the LRPD test. In [69, 70], Rauchwerger and Padua propose the use of LRPD tests in conjunction with speculative parallelization of loops, with support for backtracking and re-execution of the loop serially if the run-time test fails. Applying this method on parallel loops, they show that it usually leads to superior performance relative to the inspector-based method. The proposal speculatively executes a loop as a DOALL and apply a fully parallel data dependence test to determine if it had any cross-iteration dependences. For a given loop, the LRPD test is performed on each shared variable whose references can produce a dependence violation by creating corresponding *shadow arrays* to track read and write accesses. These shadow arrays are explored at the end of the parallel execution of the loop to detect if a dependence violation has appeared. In that case, the execution fails and the loop is re-executed sequentially. Otherwise, the loop has been executed in parallel successfully. Before checking the validity at run-time the loop is transformed through privatization and reduction parallelization.

Later, Gupta and Nim in [57] proposed a set of new run-time tests for speculative parallelization of loops that defy parallelization based on static analysis alone. On their paper they present a method for speculative array privatization that is more efficient than previous methods, and does not require rolling back the computation in case the variable is found not to be privatizable. Also, they present a technique that enables early detection of loop-carried dependences, and another one that detects a hazard to parallelization immediately after it occurs.

In [39] Hammond, Willey and Olukotun purpose a software/hardware scheme that includes a hardware support for data speculation on memory accesses that makes the parallelization of C programs. Their approach divides programs into threads and distributes the resulting threads among processors in the chip multiprocessor. Their hardware support is a speculation coprocessor which helps execute a set of software speculation exception handlers.

Rundberg and Stenström at their work [71] applied in software many of the ideas of hardware-based speculative parallelization. They contribute with a design of a software-based speculation system that reduces the overhead of the current schemes. First, name dependences are resolved by dynamically renaming data at run time. Second, the overhead of restoring system state is greatly reduced by (i) reducing the amount of state to commit and by (ii) supporting parallel implementations of the commit phase. Third, some true data dependence violations are avoided by supporting lazy forwarding without the need for enforcing synchronizations between a pair of conflicting threads. Finally, true data dependence violations are detected when they happen which can cut the cost of mis-speculations. In their approach, loads and stores whose addresses can not be disambiguated statically are associated three tasks: To aid in dependence resolution and

violation detection, all data structures that could suffer a dependence violation are associated with a support data structure. Also, each speculative instruction is augmented with checking code that detects data dependence violations dynamically. Finally, it remembers the execution order to reflects sequential semantics.

In the paper of Dang and Rauchwerger [16] a technique to extract the maximum available parallelism from a partially parallel loop is presented. This solution removes the limitations of previous techniques, i.e., it explains an extension of LRPD test (called Recursive LRPD test) that can be applied to any loop and requires less memory overhead. They propose to transform a partially parallel loop into a sequence of fully parallel loops. At each stage, they speculatively execute all remaining iterations in parallel and the LRPD test is applied to detect the potential dependences. All correctly executed iterations (those before the first detected dependence) are committed. On the other hand, those iterations that previously caused dependence violations now not have to be re-executed sequentially because authors affirm that only iterations larger or equal to the earliest sink of any dependence arc need to be re-executed, so only the remained of the work needs to be re-executed, as opposed to the original LRPD test. They re-apply the LRPD test recursively on the remaining processors, until all processors have correctly finished their work.

## 2.2 Software-based TLS tools

Cintra and Llanos [10] contribute with an scheme with an aggressive sliding window, with checks for data dependence violations on speculative stores with reduced synchronization constraints, and with fine-tuned data structures. At [11], the same authors perform a more complete evaluation of that scheme. This architecture of speculative parallelization is described in depth in Chapter 3.

An improvement of the mentioned architecture would be found in the paper of Tineo, Cintra and Llanos [76]. This is the most similar approach to the one developed in our previous work [23, 24]. In this poster, the authors introduces a pointer-based schema, i.e., a TLS support that allow the use of pointer-addressed references based on the use of several “heap” structures that keep data information.

Ramaseshan and Mueller in [68] describe their own software approximation. They are centered on the parallelization of scientific codes that tend to be dominated by regular and predictable access patterns.

Jang and Lim [82] show an architecture description specially designed to scalar applications. Also contains the design of a specific compiler specially implemented to the mentioned architecture. So, it combines hardware and software approximations. The architecture is composed by two cores: One to execute the main program thread, and the other to execute speculative threads. Threads of each processor have their own registers. On the other hand, the compiler developed only selects those loops that are likely to improve delivered performance.

Kelsey, Bai, Ding and Zhang developed a system called *FastTrack* that performs an unsafe optimization of sequential code [46], i.e., they created a software system that manages speculative parallelization. Specifically, their programming interface enables programming by suggestions, so, user can suggest faster implementations based on partial knowledge about a program and its usage. They divide code in two branches, the fast

track and the normal track, and programmers can change between both tracks when they want.

In 2008, Tian, Feng, Nagarajan and Gupta in [74] proposed the Copy-or-Discard (CorD) execution model, in which the state of speculative parallel threads is maintained separately from the non-speculative computation state. If speculation is successful, the results of the speculative computation are committed by copying them into the non-speculative state. If misspeculation is detected, no costly state recovery mechanisms are needed as the speculative state can be simply discarded. Some years later, Tian, Feng and Gupta developed mechanisms that enable CorD to efficiently supports speculative execution of programs that operate on heap based linked dynamic data structures. Their are described in [73]. In particular, they proposed a copy-on-write scheme which limits the copying to only those nodes in the dynamic data structure that are modified by the speculative computation. When a speculative thread writes to a node in a dynamic data structure for the first time, the node is copied into speculative state. If a speculative thread only reads a node in the non-speculative state, it is allowed to directly access the node from the non-speculative state.

Another work of Tian, Lin, Feng and Gupta is [75]. In this paper they developed an approach for incremental recovery in which, instead of discarding all of the results and re-executing the speculative computation in its entirety, the computation is restarted from the earliest point at which a mis-speculation causing value is read. With those advances the cost of recovery is reduced as only part of the computation is re-executed, and, since recovery takes less time, the likelihood of future mis-speculations is reduced. The main idea to decouple the space allocation from thread creation is to create a new subspace when a speculate value is first read.

Feng, Gupta and Neamtiu developed a system that deals with efficient parallelization of hybrid loops in [26], that is, those loops that contain a mix of computation and I/O operations. They tried to get that purpose by applying DOALL parallelism to hybrid loops by breaking the cross-iteration dependences caused by I/O operations. Authors developed a support to enable speculative parallelization of hybrid loops by performing some modifications to the code. To effectively parallelize hybrid loops they developed techniques for reducing bus contention, specifically, they proposed the use of helper threading.

An adaptive approach for speculative loop execution that handles nested loops has been recently proposed [27].

## 2.3 Hardware-based TLS tools

Several hardware implementations have been developed to support TLS. The major change that has to be done in a conventional processor is the addition of some auxiliary registers that manages speculation.

Speculative Versioning Cache is the system proposed by Gopal et *al.* in [37]. This approach supports speculative execution of applications through the use of a cache-based architecture. Hence, each processor has its own cache to prevent bottlenecks.

Steffan et *al.* in [72] propose a hardware TLS approach whose goals are (a) handle arbitrary memory access patterns, because they affirm that previous works only could use array references; and (b) provide a scalable architecture, i.e., a system usable for all the

processors of the time.

Cintra, Martínez and Torrellas [13] designed and evaluated a scheme for scalable speculative parallelization that requires relatively simple hardware and is efficiently integrated next to the cache coherence protocol of a conventional NUMA (Non-Uniform Memory Access) multiprocessor. They have taken a hierarchical approach that largely abstracts away the internals of the node architecture. In particular, they were able to utilize a self-contained speculative chip multiprocessor as building block, with minimal additions to interface with the rest of the system. The integration of speculative chip multiprocessors into scalable systems seemed to offer great potential. Their scheme include a multi-ported table called MDT that records speculative accesses to data. Specifically, as memory lines are being accessed, the MDT allocates a Load and Store bit per word in the line and per processor. Later, when another processor loads a word, the Store bits are used to identify the most updated version among the predecessors. When a store operation is performed, both bits are used to detect premature loads by successor threads. Another architecture specially designed to support TLS requirements and manage dependences can be found in the work of Barroso et *al.* [6].

The Hydra chip multiprocessor (CMP) [38, 61] was the Stanford hardware approximation. Several researches have been developed with this architecture, for example, in [62], Prabhu and Olukotun use it to perform a manual TLS parallelization of SPEC2000 benchmarks. With this work they affirm that TLS support should be added to future CMPs.

Nowadays, Intel's Haswell processor support Hardware lock elision, a hardware mechanism that enable to execute on parallel sequential codes with the use of transactions. To achieve this task, this new processor introduces two transactional synchronization extensions to the x86 architecture [2]. In spite of the recently launch, several approaches have subjected some improvements to this extensions, such as Afek et *al.* in [3].

## 2.4 TLS combined with other parallelism techniques

Hammond et *al.* affirm that in cluster systems, most of the conventional speculative schemas guarantee only single-threaded atomicity [40]. Therefore, Garzaran et *al.* affirm in [33] that TLS can not always improve the speedup obtained.

Nowadays, some solutions based on combine parallelization models with others are developed. Many solutions use transactional memory (TM) model combined with others. TM is based on dividing code into transactions and then execute them in parallel. This technique shares most of the semantics of TLS, however in TM threads do not have to maintain an order. In this context, Vachharajani et *al.* [77] introduce Multi-threaded Transactions (MTXs), where speculative work done in different pipeline stages, i.e., by different threads, can be committed together. In this work, its authors designed a solution based on changes to the cache coherence protocol. Raman et *al.* in [66], propose another idea to support the mentioned combined parallelism with TM that uses a hardware with cache-coherent shared memory. In addition, Mehrara et *al.* [58] describe *STMLite*, a software transactional memory model modified to support speculative parallelization. Furthermore, Kim et *al.* in [47], combine pipeline parallelism, speculative pipeline parallelism (a technique that requires MTX), and TLS.

In [86], Zhao, Wu and Shen proposed the use of probabilistic analysis into the design of speculation schemes. In particular, they focused on applications that are based in Finite-State Machine. Authors affirm that this type of applications have the most prevalent dependences among all programs, however they show that the obstacles for effective speculation can be much better handled with rigor. They developed a probabilistic model to formulate the relations between speculative executions and the properties of the target computation and inputs. Based on the formulation, they proposed two model-based speculation schemes that automatically customize themselves with the best configurations for a given Finite-State Machine and its inputs.

## 2.5 TLS classification

In [45], Kejariwal et *al.* classify TLS into three different types: (1) control speculation, (2) data dependence speculation, and (3) data values speculation. These types are not disjoint, and their basis could be combined to achieve better results.

Control speculation (CS) applies speculation to some of the loops with conditionals. Each iteration detects its execution path and then they are mapped on to different threads. In this kind of TLS a thread detects a control violation when it is not executed at all. [64] is an example of this speculation because Puiggali et *al.* tried to predict conditional branches that would be followed without knowledge about all the variables implied in the condition.

Second, data dependence speculation (DDS) leads with those loops that have inter-thread memory dependences. In these cases values are optimistically predicted and discarded if they fail.

Finally, data value speculation (DVS) predicts at run-time the result of instructions before they are executed. This point of view performs a *value prediction*. This approximation is based on the idea that values of the iterations to execute are predicted to be executed and avoid squashes. For example, Raman et *al.* in [67], describe a prediction-based TLS software that predicts values of nearer iterations without specify the iteration where a value will be token. The main disadvantage of these approximations is that in general, for applications loops with irregular memory accesses and complex control flow, they do not obtain good predictions.

## 2.6 Applications behaviour under speculative parallelization

Kejariwal et *al.* perform an analysis of the thread speculation using the benchmark SPEC CPU2006 [44] and affirm that the use of TLS with these benchmarks has no benefit. Hertzberg and Olukotun [41] also uses these benchmarks to present its technique based on speculate over running applications to extract parallelism called Runtime Automatic Speculative Parallelization (RASP).

Exists some out-of-order engines that try to extract parallelism of sequential programs with the use of a look-ahead guide that examine “future instructions” of the program to examine them and then perform a parallel execution. However, several times the guide

itself is the main bottleneck of the program. In [32] Garg et *al.* use TLS to avoid some of the mentioned overheads.

## 2.7 Design choices in TLS

The main designs choices that can be done in a TLS system are well described by Yiapanis et *al.* in [83]. There are several ways to implement a TLS, and the choice depends on the target applications:

- **Concurrency control**: This term refers to how are the conflicts treated when they are detected. We can locate two different types:
  - *Pessimistic Concurrency Control*: With this approach when a conflict is detected, it must be resolved immediately. Speculative threads only can access to their own locations and an underlying system checks for dependence violations.
  - *Optimistic Concurrency Control*: With this approach speculative threads can access to the same location at the same time, and conflicts are detected and resolved in a later stage.

  Yiapanis et *al.* [83] affirm that Optimistic Concurrency Control is normally better than the other approach when conflicts are expected to be rare.

- **Version Management**: To maintain A TLS system, some additional data have to be managed to preserve the semantics. So, this term refers to the way that data versions are managed by the TLS system. We can distinguish two kind of approaches:
  - *Lazy Version Management*: Threads that use this idea maintain their own local copy of the data managed, therefore, when a load or store operation is performed only the local version is changed. So, if a conflict is detected, only the local version of the thread that are in conflict has to be discarded, instead of the reference version in memory.
  - *Eager Version Management*: In this context, threads modify directly reference version in memory. So a buffer (called *undo log* in the literature) that records old values is needed to support rollback operations.

- **Conflict Detection**: This concept refers to how strict will be the system at time to locate dependence violations. Again, there are two types of conflict detection:
  - *Lazy Conflict Detection*: This approach avoids the need of check for conflict on every access. This task is delayed to a later stage before the commit operation.
  - *Eager Conflict Detection*: This approach looks for conflicts on every access to avoid wasting time in operations performed when a conflict has happened.

- **Scheduling**: This term refers to how the loop is partitioned and assigned to threads in a speculative execution. There are two ideas:

  - *Static scheduling*: Iterations are uniformly distributed along threads and chunks are assigned to them statically.
  - *Dynamic scheduling*: Here, chunks are set with a different number of iterations, and are assigned at threads at runtime.

  An additional type is described by Cintra and Llanos in [11] that distributes chunks of iterations along slots of a sliding window.

- **Squashing alternatives**: This concept, addressed by Garcia-Yaguez et *al.* in [28, 30], refers to the fact that when a dependence violation is located, thread should be discarded. Some approaches discard threads that consumed the wrong value, and some other all successor threads. The possible behaviours that could arise are:

  - *Stops parallel execution*: The first solutions simply discard the speculation when a violation appears and restart the loop serially. With this approach only those loops that do not have inter-dependences could be benefited.
  - *Inclusive squashing*: This approach stops and restarts the first thread that manages the wrong value and all its successors.
  - *Exclusive squashing*: Only offending threads and those successors that have consumed any value generated by them are discarded and restarted. In [28] García, Llanos and González-Escribano describe a software-based solution that purposes an *exclusive squashing* mechanism, where only offending threads and all their successors that have consumed any value generated by them are discarded. Their solution is based on solution mentioned in [10, 11], and for their purpose have handled a list that contains dependences between producer and consumer threads. Experimental results shows that this solution usually improves execution times of the previous version of the speculative parallelization engine used.
  - *Perfect squashing*: Discard offended threads and those successors that have consumed any wrong value. This is clearly the best approach, however, to the best of our knowledge, there are not any implementations of it, mainly because an in-depth analysis should be performed to detect the wrong values, and this operation is too costly.

A more in depth description would be found at [28, 30, 83].

Besides this, the main contributions located at [83] are (a) the introduction of a new structure that optimize memory overheads of classical approaches based on the idea of mapping every user-accessed address into an array of integers using a hash function, (b) the implementation of a speculative library with the mentioned structure based on a Eager Version Management design called *MiniTLS*, and (c) another implementation based on Lazy Version Management design called *Later* that uses a combination of inspector-executor [49] and LPD [16, 69, 70] techniques. Experimental results are compared to the speculative tool developed by Oancea et *al.* [60], a system whose main design principle is to decrease overheads of speculative operations.

Finally, [29] presents an speculative architecture that manages exceptions that would appear within speculative executions.

## 2.8 TLS as a help to manual parallelization

In order to avoid making speculative codes, that were slower than the original sequential codes, some researches have proposed techniques to predict overheads of speculative parallelization. For example, the work developed by Dou and Cintra in [20] contains a compiler that can be used to estimate the overheads and expected resulting performance gains, or losses. Later, in [21], the same authors give a more detailed and optimized version of this work.

Ding et *al.* [19] propose a software-based TLS system 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 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 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 [43], Ke et *al.* propose a system that rely on dependence hints provided by the programmer to allow explicit data communication between threads, thus reducing runtime dependence violations. Xekalakis, Ioannou and Cintra in [81], propose a model that combines different techniques such as thread-level speculation, helper threads and run-ahead execution, in order to dynamically choose at runtime the most appropriate combination.

Zhang et *al.* [84] describe continuous speculation, a new technique whose mainly objective is to achieve that no processor are not busy, i.e., each thread will have always a task to do. For that purpose it uses speculation based on process to achieve parallelization of large sequential codes. Its solution uses a sliding window and a group classification to ensure correct order of the task. To get information about the possibly parallel regions of a sequential code, it uses BOP, the tool described in [19].

Some authors have focused on provide assistance to those programmers that extract TLS of the applications. For example, [4, 5, 80] show tools that make an analysis of the codes to give information about the best loop to be speculative parallelized. More specifically, Wu, Kejariwal and Caşcaval developed a compiler that set a serie of labels that allow the design of a profiler to ease the detection of dependence at run-time [80]. Aldea in [4, 5] describes a different process that use XML capabilities to achieve the best possible degree of parallelism. The main goal of this work is to achieve an automatic speculative parallelization of applications. In this paper, a software is described that divides process, mainly in several steps: First, a loop that is able to be parallelized is located, then a tool (based on Cetus compiler) generates an XML tree based on the internal representation of the source code. After that, another tool called Loopest is used to perform an analysis on variables usage and loops to augment the XML tree with the code needed for speculative execution. Finally, the resulting XML tree is translated back to C code using another tool called Sirius.

Another type of compilers and helper tools for TLS cited by Wu et *al.* in [80] are [22, 42, 51, 65, 78]. Another type of dependence profiler has been designed by Chen et *al.* and could be found at [9].

### 2.8.1 Automatic speculative library

There are also libraries that perform speculative parallelization directly, such as in [63], where Prabhu, Ramalingam and Vaswani have developed some directives and operations to allow programmers to make their own speculative programs. Authors proposed two language constructs, called speculative composition and speculative iteration to enable programmers to declaratively express speculative parallelism in programs. They presented a formal operational semantics for the language to define the notion of a correct speculative execution as one that is equivalent to a non-speculative execution. Using mentioned semantics, they described a set of conditions under which such rollback can be avoided with the corresponding time reduction.

## 2.9 TLS to GPU

Nowadays, parallelism applied to GPUs is one of the major ways of research in that field because the possibility of use many more processors. This fact makes that apply speculative techniques to that kind of parallelism is, at least, desirable. Liu, Eisenbeis and Gaudiot discusse how TLS could be correctly used in the parallelism of GPU [50]. In this way, in a recent approach, Zhang et *al.* introduces a new library based on sliding windows that support TLS in GPUs [85]. To the best of our knowledge, this is the first schema that implements this techniques, and for that purpose, authors have adopted their software with ideas of the classical solutions that are expected to have a better behaviour over GPUs, for example, using a hybrid dependence checking, or a parallel commit scheme.

# Chapter 3

# Original speculative library

*La librería software de paralelización especulativa usada como base se describe en [10, 11]. Por tanto, y con el objetivo de clarificar la visión acerca del motor de paralelización especulativa descrito, se facilita una introducción y una descripción de su funcionamiento. Se mencionarán las limitaciones que conlleva el uso de este software: (a) debemos conocer las iteraciones del bucle sobre el que se especulará, (b) no se trabajará con aritmética de punteros, y (c) no se utilizará memoria dinámica. Por otra parte esa librería obliga a realizar una clasificación manual de las variables, ya que el modelo se basa en la librería de paralelización OpenMP [1].*

*Además cabe mencionar que la estructura sobre la que se basa el funcionamiento de este motor especulativo es una ventana deslizante cuyos slots son asignados a threads que ejecutan sus iteraciones.*

*Por último, también cabe decir que el uso de esta librería conlleva que se cambien manualmente todas las lecturas sobre variables especulativas por la función "specload", y todas las escrituras por la función "specstore". Adicionalmente para consolidar los datos, debemos establecer una llamada a la función "threadend".*

*Otra de las funcionalidades descritas es la paralelización de las llamadas operaciones de reducción como por ejemplo el cálculo del máximo; o las funciones de inicialización del sistema.*

```
1 for (i=1; i<5; i++)
2 {
3   LocalVar1 = SV[x];
4   SV[x] = LocalVar2;
5 }
```

Figure 3.1: Example where appear a RAW dependence. This type of loop may cause dependence violations in parallel executions.

Software-based speculative parallelization libraries aim to achieve a concurrent execution of the loops with cross-dependences in their iterations. The library that is going to be introduced here was the origin of our work, so, it is worthwhile to describe it in depth.

This engine [10, 11, 53, 54] is based on a set of function libraries that allows the speculative execution of a loop. It is a work-in-progress that requires the programmer to add additional code to parallelize it. This process is expected to be automatic in the near future [4, 5].

In this way, let us see an example of the speculative execution of a loop with the mentioned tool. Our example is based on loop of the Figure 3.1 with four iterations. Suppose that our system has enough processors to execute a single iteration in each one. At this time could be mentioned that thread of the lesser iteration is called non-speculative thread, and thread of the highest iteration is called most-speculative (these concepts would be deeper exposed throughout this chapter).

Each thread has its own version of shared data, so, if each thread commits its results with no order, some incoherences at the end of loop execution may appear. Therefore, each thread should commit its data in order.

To have a better understanding of this system, an example of the execution of a loop its shown at Figure 3.2. This figure shows a possible parallel execution of the loop depicted in the example of the Figure 3.1. All operations are done maintaining sequential semantic until instant **t10**. At this moment, thread three modifies the value that belongs to the shared vector $SV[X]$ used at **t7** by thread four, promoting that results from thread four calculated so far are discarded.

To avoid undesirable failures some modifications in original source code are needed. Those modifications are depicted at Figure 3.3, and are the following.

- **Load operations** over speculative variables are replaced with a function that recovers most recently value, that is, the most updated value of the variable.
- **Store operations** over speculative variables are replaced with a function that stores the value and detect sporadic dependence violations.
- Each thread executes a **commitment of calculated data** at the end of the execution of its chunk of iterations. Also, this operation should assign a new chunk of iterations to be executed.

However, this version of the engine has got some requirements:

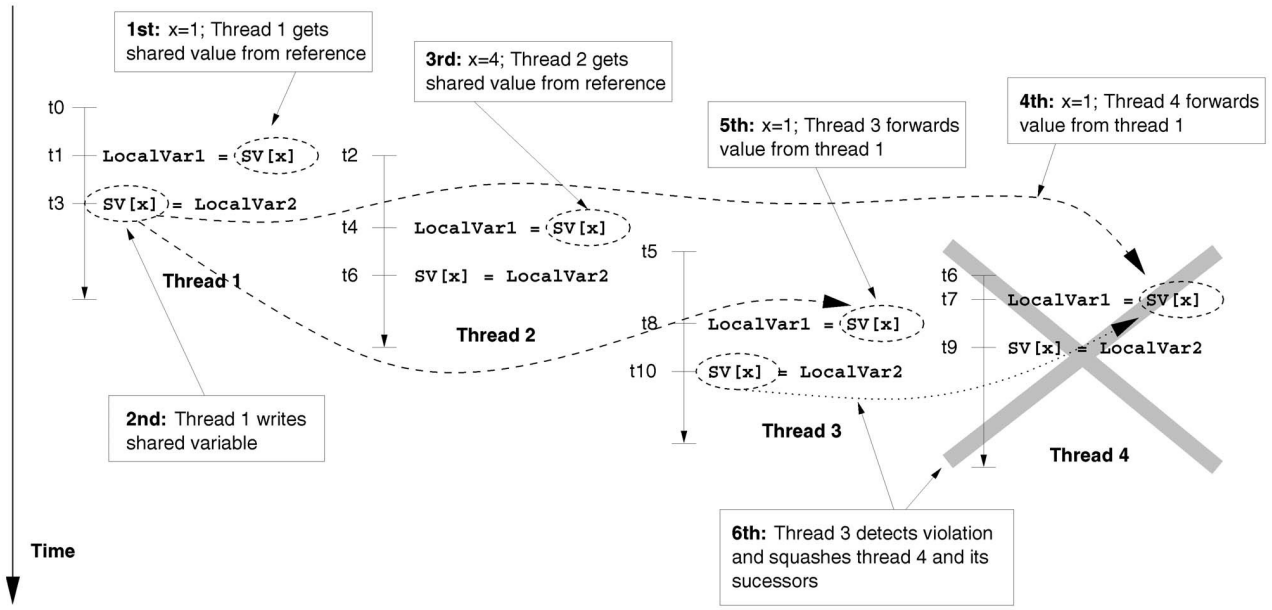



Figure 3.2: In each speculative load operation is searched the most updated version of $SV[X]$, consulting previous threads until arrive to non-speculative thread. At each speculative store operation is checked if any following thread have used a wrong value, and then, if is appropriated its execution is discarded.

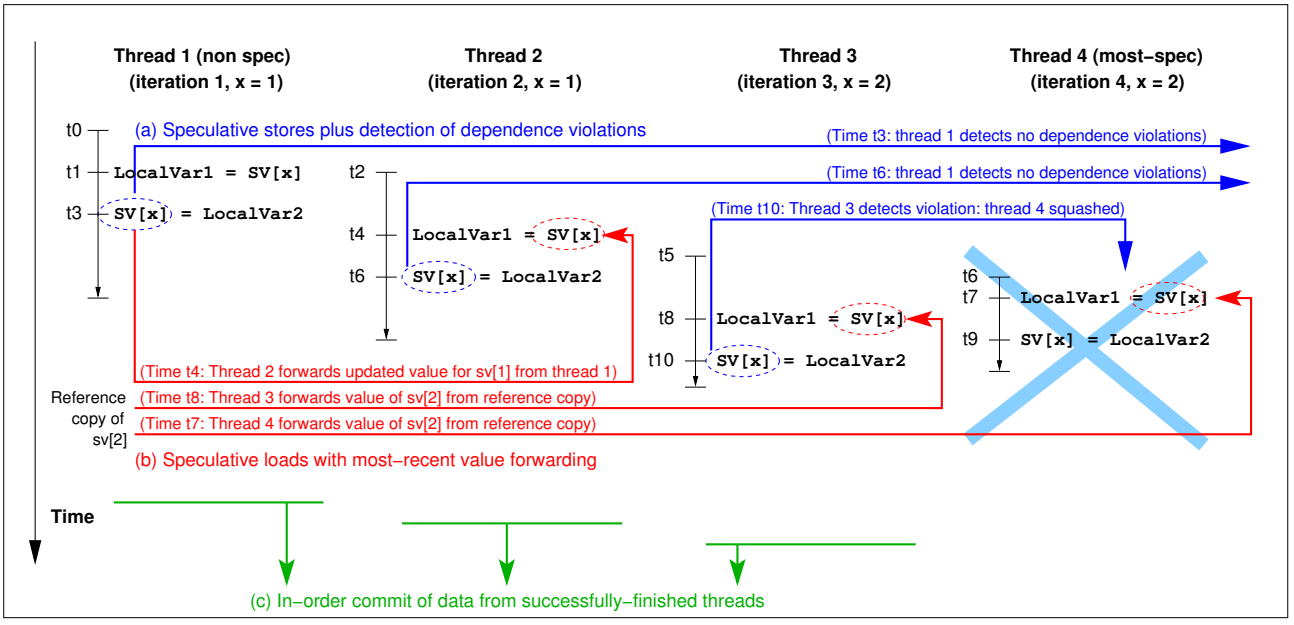



Figure 3.3: On the one hand, speculative store operations perform data reviews looking for matches with data of its successors. On the other hand, speculative load operations try to find the datum on demand in the previous threads, i.e., look for this datum in the lesser speculative threads. Commit operation should be performed maintaining the sequential semantic.

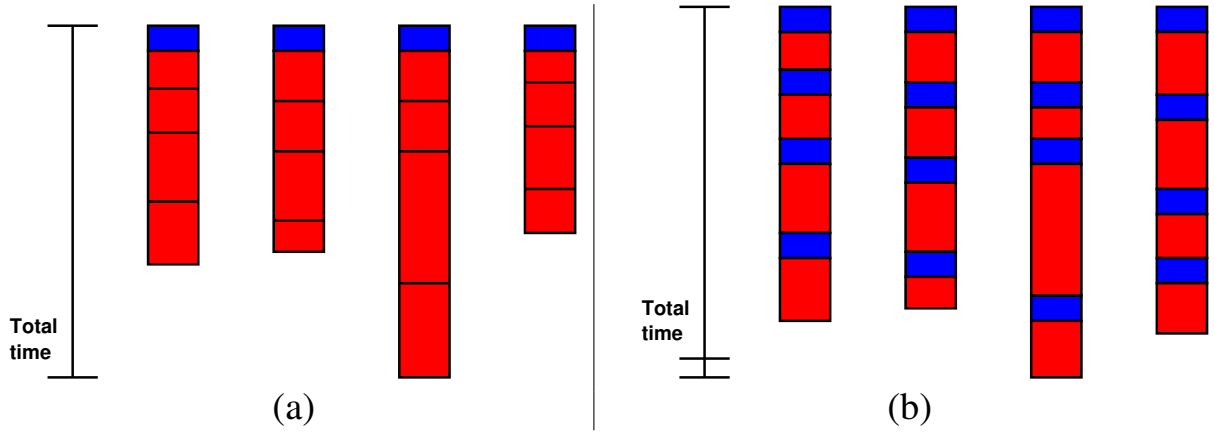



Figure 3.4: Block comparison: (a) Chunks with a few iterations, and (b) chunks with some more iterations.

- Analyzed loop should have a **number of iterations known** before its execution.
- Analyzed loop **must not work with pointer arithmetic**.
- Inside analyzed loop **can not be used dynamic memory**.

Once accomplished those premises, a **manual, previous analysis** of the execution of the loop classifying all the available variables is required. In this way, the user labels a variable as **private** when it is always stored before its use in the same iteration. Label of **shared** variable is given to those variables that are always loaded. Rest of variables, labeled as **speculative**, are in risk to suffer a dependence violation.

In addition to the mentioned requirements (modify the mentioned operations and perform a variable classification), some changes are needed in the **original loop structure**. Suppose that the loop to be parallelized has $N$ iterations and $P$ processors are available to perform the execution. Parallelization begins by replacing original loop with a `doall` from 1 to $P$, and then each processor is assigned with a chunk of consecutive iterations. When a thread ends its chunk, calls a function that assign a new one. When no more chunks could be assigned, and all threads end their execution, could be said that execution is ended.

The last point referred to the original speculative library is about the size of blocks of iterations. In this way, and in spite of wasting more time with the assignation of new blocks, is highly recommended to use blocks of few iterations. If we use blocks of many iterations, the global execution of the program would be delayed, as it can be seen in Figure 3.4.

## 3.1 Architecture of this speculative engine

This speculative schema aims to avoid dependence violations with the use of several data structures.

### 3.1.1 Auxiliary data structures

A data structure is needed to save thread states. To that purpose, a *sliding window* is used with a similar, or higher, size than the number of processors, i.e., if we have $W$ slots and $P$ processors $W \geq P$. Each slot contains its own data version because each thread should have its own version of speculative data, so, each thread needs a vector to save and commit, or discard, its data. In this sense, a global vector where perform the final commitment of data after the execution is needed too. At least as much vectors as available processors should exist, i.e., it should be $W$ auxiliary vectors besides an additional global vector: $W + 1$ vectors with $M$ items (where $M$ is the number of speculative variables).

Each slot of the mentioned *sliding window* has a state that holds the current situation of this slot. State possible values are the following:

- `FREE`: Slot is free and could be assigned to a thread. At the beginning all slots of the *sliding window* are free.
- `RUNNING`: Slot is busy, that is, executing some iterations.
- `PENDING_SQUASH`: This value appears when a wrong value is detected by a thread in one of its speculative variables, and therefore execution of this thread should be discarded.
- `SQUASHED`: Slot is busy by a thread that should start its execution again because a dependence violation has been detected.
- `DONE`: Slot has successfully ended its execution.

*Sliding window* allows to assign consecutive threads until complete `P` available processors, when those `P` slots start their execution (`RUN` state). When a thread finishes its execution, the window is moved to the right, therefore, window should also contain two indicators: An indicator to the non-speculative slot, and another to the most-speculative one. The structures that have been described are depicted at Figure 3.5.

We have seen that each slot is assigned to a processor, and has associated a version of speculative variables. Moreover, threads should know the state of each speculative variable, that is, if that variable has been loaded, modified, etc. So, each slot has an *Access Matrix* with `M` positions. Each position of this matrix contains the state of each variable version. Available states are the following:

- `NotAcc`: The element has not been accessed by this thread. This the initial state of each element.
- `ExpLd`: *Exposed Loaded*, the thread has read the value of the variable of this element.
- `Update`: The thread has modified value of the variable of this element.
- `ElUp`: This state appears when a thread that loaded the value of the variable, modifies this value.
- `RedAdd`: A reduction operation in the sum of this element has been applied.

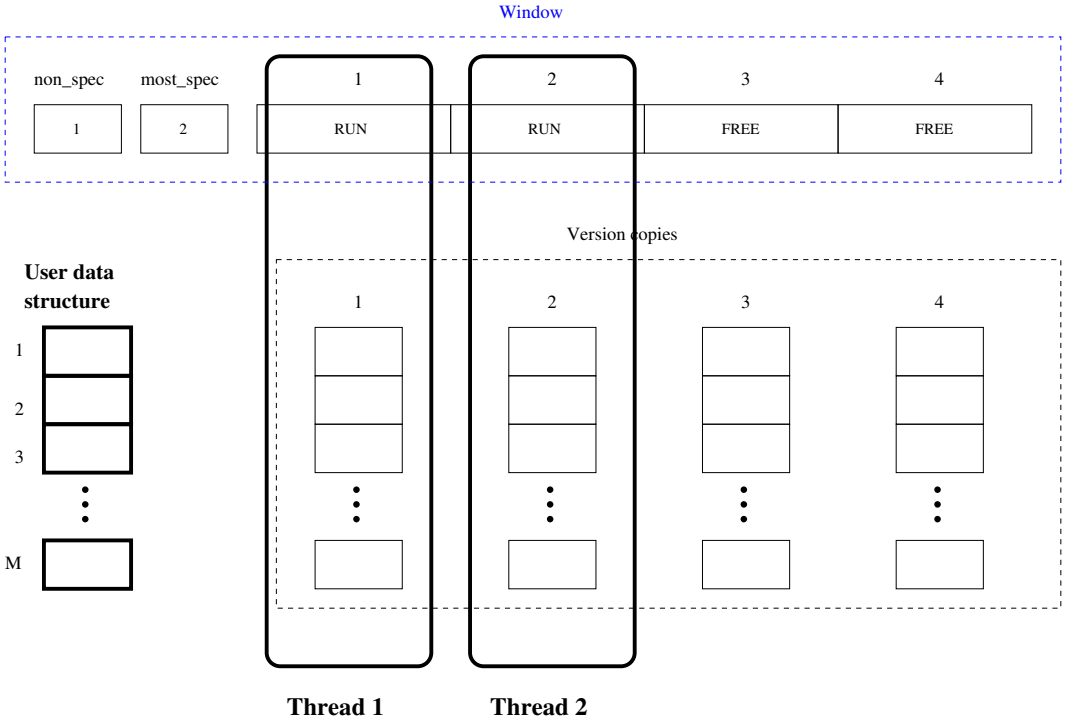



Figure 3.5: Main elements of the original speculative engine

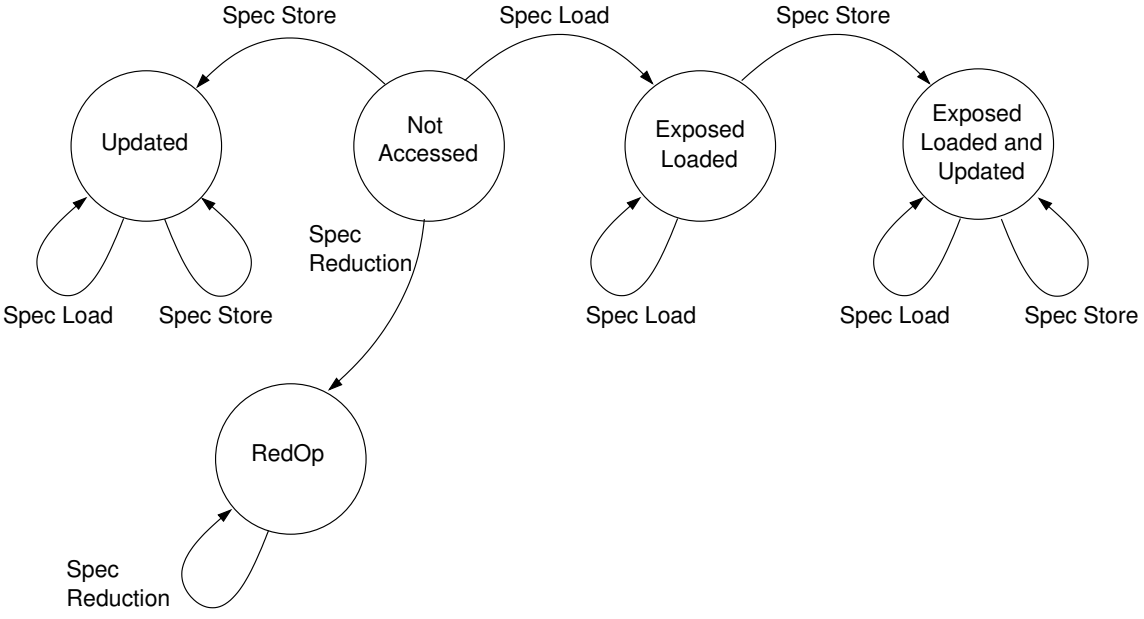



Figure 3.6: Evolution of the possible states that each element could achieve in the Access Matrix

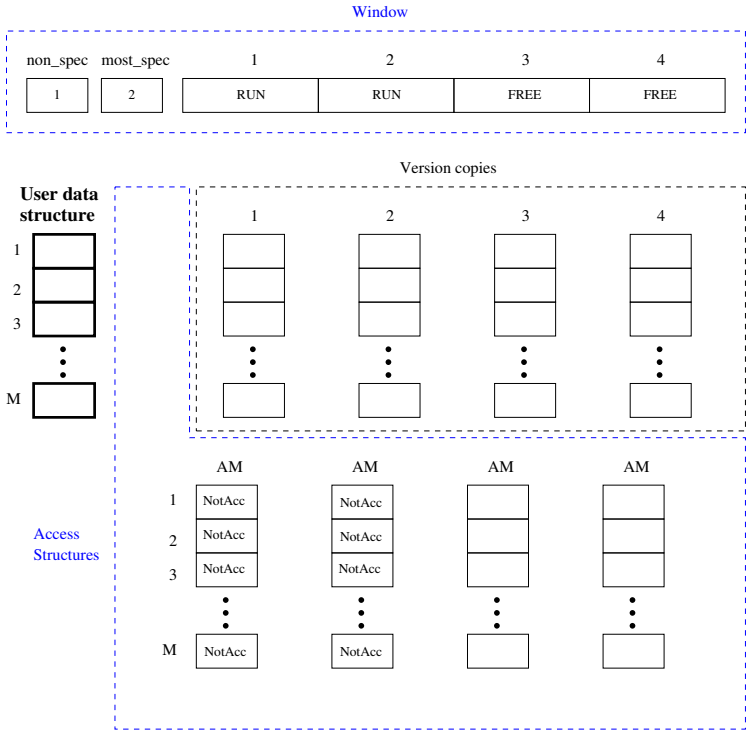



Figure 3.7: Original architecture of the speculative engine.

- `RedMax`: A reduction operation in the maximum of this element has been applied.

Figure 3.6 shows a automata whose states are those that could be stored in the elements of the *Access Matrix*. These states evolve through load or store operations.

A scheme of architecture described is shown at Figure 3.7.

## 3.2 Speculative load operations

At compile time, when a speculative variable is going to be read, this instruction should be replaced following the next way:

```
LocalVar = maincopy[index]
```

⇓

**specload(index, current, LocalVar, maincopy, myIVtail)**

Arguments have the following meaning:

1. `index`: Element position in the speculative vector.
2. `current`: Slot number assigned.
3. `LocalVar`: Variable where loaded datum would be stored.
4. `maincopy`: Shared data structure where the element is read of the position `index`.

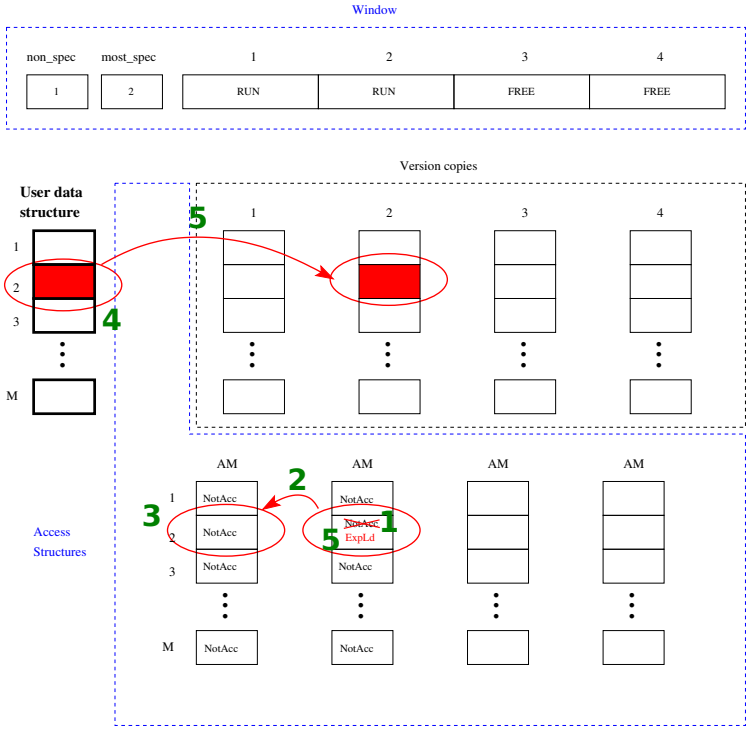



Figure 3.8: Load operation example.

5. `myIVtail`: last element accessed by the indirection vector (this structure is described where the commit operation is, in one of the following subsections).

Let us explain this operation with a generic example: Suppose that thread `N` needs to load a speculative value.

1. Thread consults the value of the position of the variable in its *Access Matrix* to obtain the most updated value. If that value is `NotAcc`, thread has not an own copy, therefore it searches in the structures of the previous threads.

2. This thread searches the value of the variable in the `N-1` thread structures.

3. If no previous thread has used this value, thread `N` would load reference value, and store it in its data version. On the other hand, if one of the threads explored have used this datum, it would be loaded.

4. After that, the corresponding value of the *Access Matrix* is updated to `ExpLd`.

Now, let us see the mentioned operations applied to a concrete example. To do so, we will use Figure 3.8. Suppose that thread 2 should execute the following instruction: $LocalVar = SV[2]$.

1. Thread 2 needs element 2, however, it has not its own copy.

2. Datum is searched in predecessors' structures.

3. Thread 1 has not used this datum.

4. No predecessor has used this datum: Thread 2 loads reference value.

5. Reference value is stored in the corresponding version of the thread 2 and the state of this element is updated to `ExpLd` (Exposed Loaded) in its Access Matrix, completing instruction $LocalVar = SV[2]$.

### 3.2.1 Early Squashing

Load operations executed by this library will be optimized to improve the performance of applications. This optimization is called *Early Squashing* and consist on perform a squash operation before the end of the chunk of iterations. To understand this operation, let us suppose that an application performs a `specstore` (store operation described at the following subsection), this operation checks the slots of the following threads. If a dependence violation is located, the slots are changed to the `SQUASHED` state, but its execution continues, until the end of the iterations. In contrast, if *Early Squashing* operation is introduced and during an execution a `specload` operation appears, thread involved checks its state and if it is `SQUASHED`, a negative value is returned and `specload` call ends. With this optimization less time is lost with the execution of unnecessary instructions, because `SQUASHED` state implies that executions results are discarded. Therefore when load operations are performed in an application should be checked `LocalVar` value in order to know if it is equal to `-1`, and in that case, *Early Squashing* operation will cause a "jump" until the part of the code where thread end the execution of its block.

## 3.3 Speculative store operations

Store operations are also replaced at compile time with a function call similarly to load operations previously described:

```
maincopy[index] = LocalVar
```

⇓

**specstore(index, current, LocalVar, myIVtail)**

Arguments have the following meaning:

1. `index`: Element position in the speculative vector.

2. `current`: Slot number assigned.

3. `LocalVar`: Variable that contains the datum that would be stored in the speculative vector.

4. `maincopy`: Shared data structure where the element is read of the position `index`.

5. `myIVtail`: last element accessed by the indirection vector (this structure is described where the commit operation is, in one of the following sections).

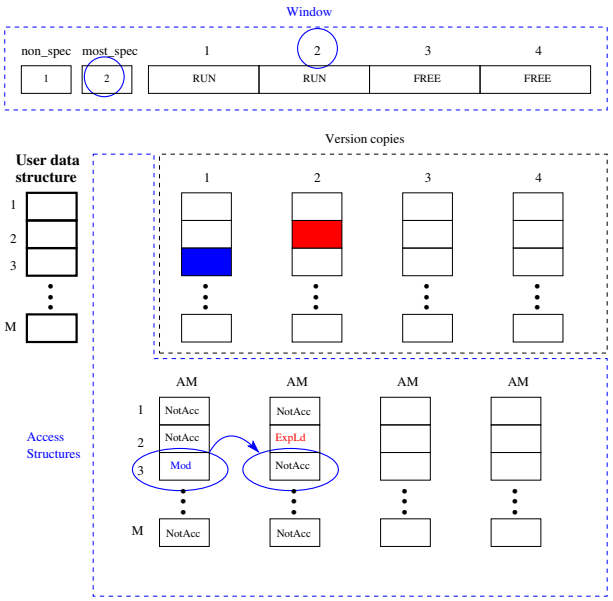



Figure 3.9: Speculative store operation description where successor threads have not used the datum to modify.

Let us expose this operation with a generic example: Suppose that thread `N` needs to store a datum into a speculative value.

1. Thread that performs this operation, modifies `index` value of the position of its own version vector.

2. The corresponding value of the *Access Matrix* is updated. Therefore if the previous state was `Not Accessed`, it is changed to `Updated`. On the other hand, if the previous state was `Exposed Loaded`, it is changed to `Exposed Loaded and Updated (ElUp)`.

3. This thread checks if any successor threads have used an old value of this datum consulting their *Access Matrices*.

   (a) If no successor thread has used this value, (see Figure 3.9) this operation ends.

   (b) If one of the threads explored have used an old, and consequently, wrong value of the datum (see Figure 3.10):

      i. Execution of the thread that has consumed the wrong value and all of its successors, are stopped.

      ii. State of these threads is changed from `RUNNING` to `SQUASHED`.

      iii. Execution of all threads with `SQUASHED` state is retried, but now old values are updated.

To get a better understanding of described operation, let's see a concrete example using Figure 3.11. Suppose that thread 1 executes $SV[2] = LocalVar$:

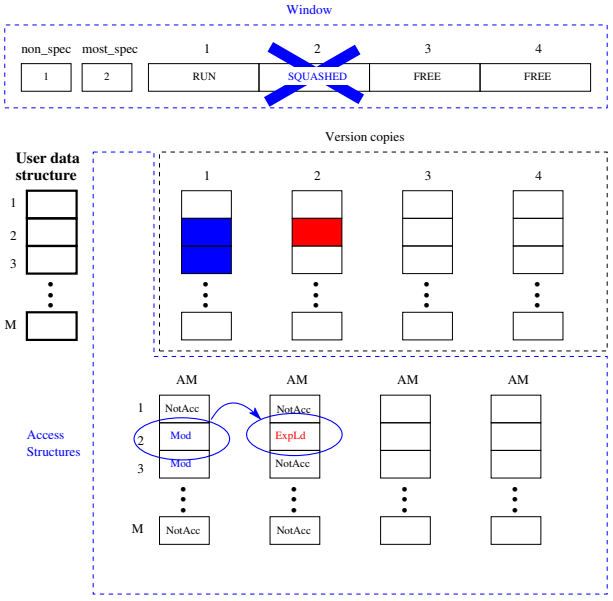



Figure 3.10: Speculative store operation description where successor threads have used the datum to modify, and consequently should be retried.

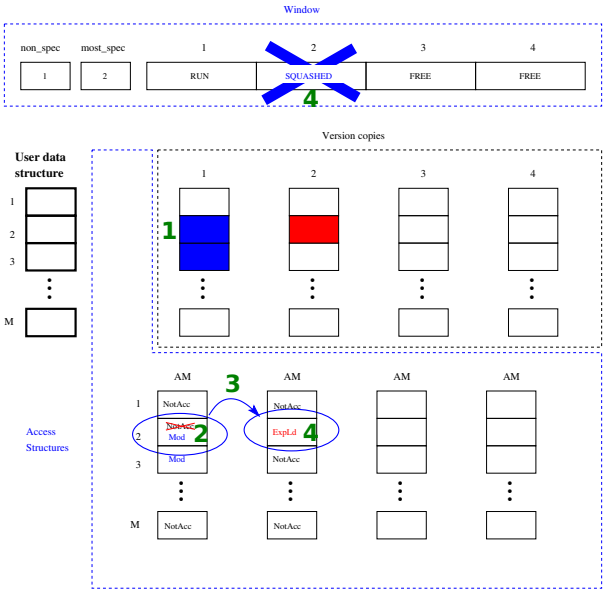



Figure 3.11: Speculative store operation example.

1. First of all, store is performed.
2. Element state was `NotAcc`, so is changed to `Mod` (if it was `ExpLd` would be changed to `ExpLdMod`).
3. Now, should be checked if a successor thread has used an old version of this datum: This situation occurs.
4. Thread that consumed the wrong datum and all its successor executions are stopped.
5. Thread 2 execution is retried, therefore, speculative load operation will load the correct value.

Speculative store operations affect only to local version of the threads, so, global vector is only changed when a commit operation is performed.

## 3.4 Results commitment

When the execution of a block of iterations ends, the state of the thread is changed to `DONE` and the function that handles commit operations is called. Tasks performed by this operation are copy values from local copies to global copy. Furthermore, function still handles the assignation of the following chunk of iterations.

When a thread ends, `commit` operation copies modified values from its own version (see Figure 3.12) to the global structure. Then the state of the slot is changed to `FREE`. If this is the first thread, `non-spec` pointer is augmented. Finally, the thread whose block has ended, is assigned to the next free slot, and then `most-spec` pointer is also augmented. Therefore, non-speculative thread changes its status from this position to the most-speculative one.

Function that implements this operation will be described in the section that introduces the optimizations performed to the commit operation.

## 3.5 Optimizations

Speculative load, and commit operations could be improved in the following way:

- **Speculative load optimization**: When a thread performs a speculative load operation all the versions of its predecessors should be checked in order to looking for the value to load. However, in the most general case data consulted by the thread have not been previously used, that is, thread should load reference value to complete the load operation. In order to avoid a unnecessary read of all the variables of predecessors, an auxiliary vector called *Global Exposed Load* is added. This vector is used to indicate if a variable has been used before, and then, when a load or store operation is performed, this vector should be updated. See Figure 3.13.
- **Commit optimization**: When a commit operation is performed all the local elements of the vector should be checked. This implies to check both modified, and

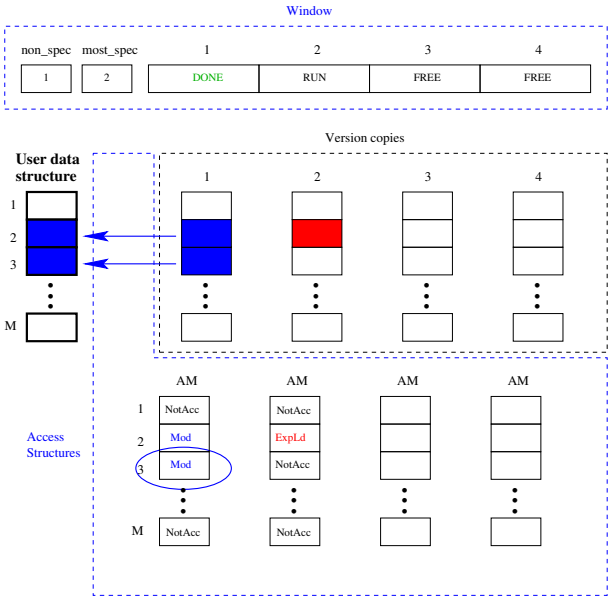



Figure 3.12: Commit operation description. Only the modified values are copied to the local version.

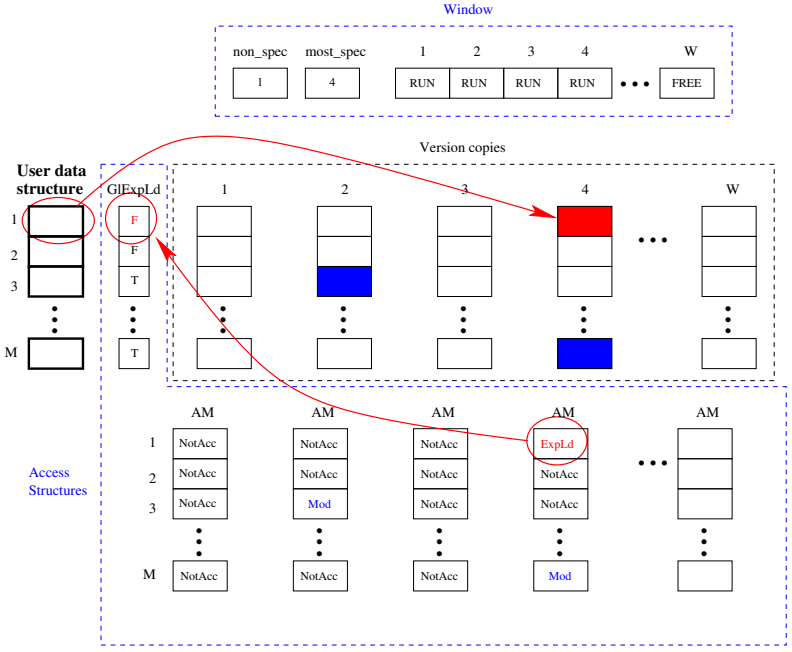



Figure 3.13: View of the use of the *Global Exposed Load* vector.

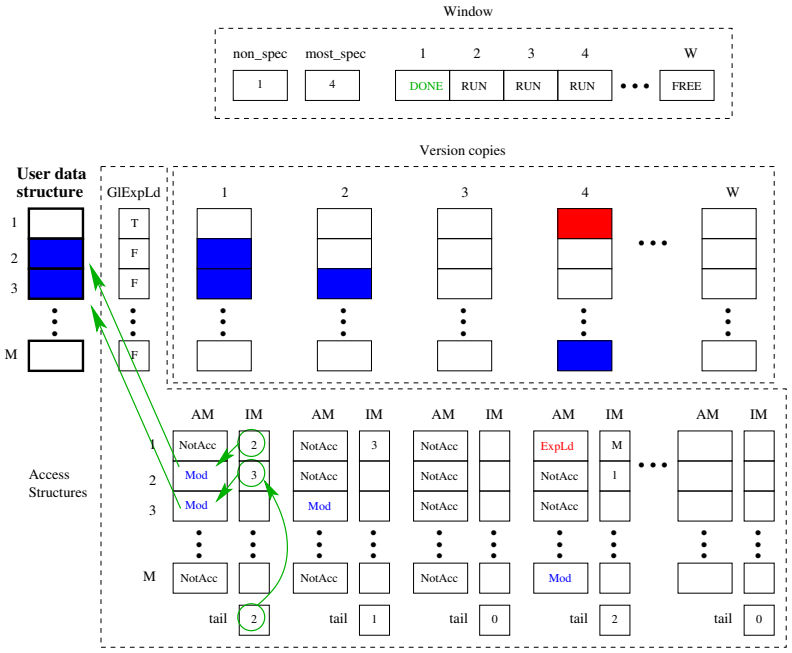



Figure 3.14: View of the use of the *Indirection Matrix* vector.

unmodified elements. We can improve this operation with the use of a list that will contain modified elements. This list, called *Indirection Matrix*, will be implemented by each thread and will save the position of the element used. Moreover, each thread will implement a variable called *tail* that will indicate the last item of the *Indirection Matrix* used in order to iterate it easier. At this moment, to commit all the values is only necessary to copy the elements of this list, instead of all local version values of the thread. Figure 3.14 shows the application of this structure.

The function that implements commit operation is called *threadend* and has the following arguments:

**threadend(current, retflag, maincopy, myIVtail)**

With the following meaning:

1. `current`: Slot number assigned.
2. `retflag`: Return value of the function. It is used to take into account if are available any more blocks. Values returned could be of two types: `JOBTODO` or `JOBDONE`. If returns `JOBTODO`, there are more blocks to be executed. On the other hand if the value is `JOBDONE`, all the iteration blocks that form the loop have been already assigned.
3. `maincopy`: shared data structure where data will be committed.
4. `myIVtail`: last busy element of the indirection matrix.

Figure 3.15 reflects the final version of this speculative engine with all the implemented structures.

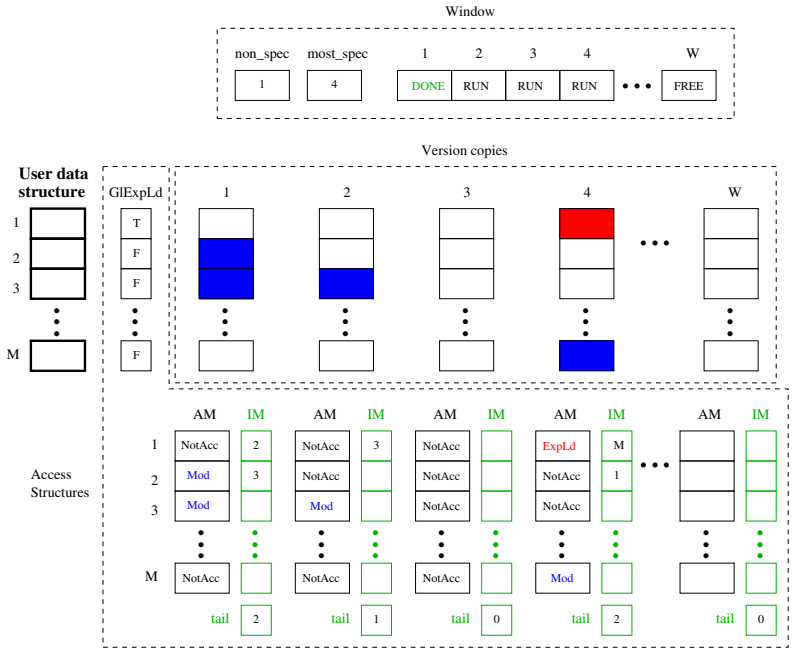



Figure 3.15: Speculative engine completed with all the auxiliary structures.

## 3.6 Reduction operations

Some operations can not be parallelized because of the own nature of the instruction. Nevertheless, in these situations, some kind of operations could be transformed in order to be able to parallelize them. When one of these cases appear, it is said that the operation can be *reduced*. For example, sum and calculation of the maximum operations can be reduced.

### 3.6.1 Sum reduction

Imagine that the loop to be parallelized contains sum operation in the following way:

**matrix(i) = matrix(i) + value**

This type of operation needs a sum operation over all the iterations of the loop. This sum operation will modify element `i` value of the `matrix`, therefore if we want to parallelize it, too many dependence violations could be produced. The main reason of this is related to the fact that iteration that modifies `matrix(i)` value needs the value of the previous iteration, i.e., `J` iteration needs to know `matrix(i)` value in the `J-1` iteration to update the value.

In order to solve this problem, a reduction operation is performed in the following way:

**matrix(i) = matrix(i) + valor1 + valor2 + ... + valorK**

In this way, instead of sum a quantity each iteration, all sums are performed at the same time in order to avoid dependence violations. This operation is known as *specadd*:

```
matrix(i) = matrix(i) + value
```

⇓

**specadd(index, current, value, myIVtail)**

Where arguments have the following meaning:

1. `index`: Position of the element in the speculative vector where the sum will be performed.
2. `current`: Slot number assigned.
3. `value`: Quantity to sum at each iteration.
4. `myIVtail`: last busy element of the indirection matrix.

This operation affects the commit operation. Now, local versions accumulate quantities at each iteration, so, if thread should execute `N` iterations, when all of them end the value will be `N*value`. At this moment, the commit operation instead of modifies the reference value, performs a sum of `N*value` each time that is called. In this way, final results are correct at the end of the operation.

### 3.6.2 Maximum reduction

Let us suppose that loop to be parallelized contains a calculation of the maximum operation in the following way:

IF **matrix(i)** < *value* THEN
**matrix(i)** = *value*

or:

**matrix(i) = MAX(matrix(i),value)**

This kind of operation can not be parallelized because dependence violations would appear continuously (as in the previous operation example). But, in the same way of the sum operation viewed, this operation could be reduced obtaining the following instructions:

**matrix(i) = MAX(matrix(i),value1,value2,...,valueK)**

At the beginning, calculation of the maximum was performed to two values, nevertheless, now is calculated over all possible quantities. This transformation allows parallelization. Function that implements this operation is the following:

```
matrix(i) = MAX(matrix(i),valor)
```

⇓

**specmax(index, current, value, myIVtail)**

Where arguments have the following meaning:

1. `index`: Position of the element in the speculative vector where the sum will be performed.
2. `current`: Slot number assigned.

3. `value`: Quantity where maximum is calculated.

4. `myIVtail`: last busy element of the indirection matrix.

This function works as follows: each thread calculate the maximum of the values of its corresponding chunk of threads. When a thread ends its execution and data should be committed, the maximum value obtained by all the threads is compared to the maximum value of the global reference vector. If value is higher than it, is copied in the corresponding position, in the other case, no operation is done.

## 3.7 Initialization functions of the engine

A previous step should be taken into account before add parallel instructions into the code. It is mandatory to initialize data structures of the engine before perform any operation. Two functions are used to perform this initialization:

- *specinit*: This function should assign the static, or dynamic, block size that slots will use.

- *specstart*: This function should initialize non-speculative and most-speculative pointers, and states of the sliding window to the `FREE` state. This operation initializes *Access* and *Indirection* matrices, those variables associated to *Indirection matrix*, and *Global Exposed Load* vector. Furthermore, the limit value of iterations is assigned at execution time with the use of the single integer argument that use this function: *specstart(int iterations)*.

## 3.8 Use of the engine and variable settings

At this moment, data structures that form speculative engine, and operations that implements are already described, so we can use it. However, some variables should be mentioned in order to complete the introduction to this library:

- *threads*: Indicate available threads to execute the problem.

- *wsize*: Window size, that is, the number of slots.

- *blk*: When block size is static, this variable implements this size, specifically, the number of iterations that form blocks.

- *shared_size*: Size of shared data structure where load and store operations would be performed.

- *cur_upper_size*: Maximum number of iterations of the loop to be parallelized.

- *max_upper_size*: Similar to *cur_upper_size*.

Once implemented the values of these constants, some modifications over the original code can be performed in order to achieve the speculative version. First of all, we have to include the file that contain mentioned variables and OpenMP library (omp_lib.h). After that, both operations to initialize data are called: *specinit* and *specstart(int iterations)*. Also another OpenMP specific function is called: *omp_set_num_threads(int threads)*, in order to indicate to the OpenMP library the number of threads that will be used. Finally, we should classify variables into two types: *private* or *shared*; this is one of the requirements of OpenMP[1, 8, 15]. Some variables used inside speculative library are always classified in the same way:

- *Private*: value, linear, current, tid, retflag, tidaux, flag, my_IV_tail, nonSpeculative.
- *Shared*: wheel_ns, wheel_ms, shadow, AV, IV, IV_tail, gELV, wheel, wheel_ol, upper_limit, varblock, jmpbuf, endLoop, endReturn.

In this point, speculative parallelization of the loop could be implemented using OpenMP directives. To perform load, store, and reduction operations over speculative variables should be replaced original instructions with the functions described along this chapter.

Finally, when the execution of a thread ends, commit operation is performed with the use of the *threadend* function, and this thread checks if there are any more blocks to be executed.

## 3.9 An example of use of this library

A step by step example developed in C language is going to be shown in order to know how to use the described engine to speculative parallelize an application. The application of the example only will have a single speculative load, and a single speculative store operations.

### 3.9.1 Sequential application

The code shown below is the sequential version of the application:

```
1  // Synthetic application written in C //
2  // Requirements: input values should be higher than 0 //
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include "myexamplevariables.h"
7
8  int main()
9  {
10
11 // Local variables
12 // P indicates current iteration, Q indicates vector index
13
14   int P, Q, aux, i;
15   FILE *file;
16   int sum=0;
17
18 // Open and read the file
19
20   if ((file = fopen("rand1000000.in", "r")) == NULL)
```

```
21      {
22        printf ( "Error opening the file \n " );
23        exit(0);
24      }
25      else
26      {
27        fscanf(file, "%d", &aux);
28        for ( i = 0 ; i < MAX ; i++)
29        {
30          vector[i] = aux;
31          fscanf(file, "%d", &aux);
32        }
33        fclose (file);
34      }
35
36    // myexamplecode: debug
37    //  for ( i = 0; i < MAX; i++)
38    //    printf("%d\n", vector[i]);
39    // myexamplecode: end debug
40
41    // Loop can not be parallelized because the index of
42    // the elements of the vector written depends on the
43    // input values of the file
44    // THIS IS THE DESIRED SITUATION
45    // LET'S START
46
47      for ( P = 1 ; P <= NITER ; P++ )
48      {
49    //  Loop code
50
51        Q = P % (MAX+1);
52
53        aux = vector[Q-1];
54
55        Q = (4*aux)%(MAX+1);
56
57        vector[Q-1] = aux;
58
59      }  // END for
60
61      printf(" Vector results \n");
62      for ( i = 0; i < MAX; i++)
63        sum = sum + vector[i];
64      printf("%d\n",sum);
65    }
```

First of all, a specific application that produces some random numbers have been developed, and a file that contains a million of random numbers is produced. In the code shown above, from line 18 to 34, file is read and random numbers are obtained. In spite of the fact that numbers are random, input file will always be the same, therefore, similar experimental results will be obtained, achieving a deterministic version of this application. Numbers of the file should be higher than 0 because they are used as the indices of a vector.

Once the file is read, and their data are stored at `vector` variable, main loop starts with `NITER` iterations, from line 47 to 59. Now, an element of the vector is read (line 53) and another one is written (line 57). `Q` variable stores the index of the element to be load or stored:

- In the case of load operation, `Q` value is equal to the remainder between current iteration and the maximum size of the vector, plus one.
- In the case of store operation `Q` value is equal to the remainder between value load multiplied by four and the maximum size of the vector, plus one.

`Q` value is unknown at compile time because it depends on the value read from the input file. Therefore, compiler can not guarantee concurrent execution of some iterations without errors, so, loop is not parallelized. In this kind of situations is when the described engine is useful, because a solution to those problems that conventional compilers can not parallelize is provided, and this algorithm is a perfect example.

Finally, from line 61 to 65 is checked if the calculated data are correct by summing `vector` elements.

It is interesting to describe the meaning of the variables used in the synthetic application, before start the explanation of the process followed to speculative parallelize this code:

- *NITER*: The number of iterations of the loop.
- *MAX*: Vector size.
- *aux*: Used to store value of an element of the vector.
- *Q*: Used to save index of the vector.
- *P*: It indicates current iteration of the loop.
- *sum*: It stores the sum of the elements of the vector at the end of the execution. It is used only to check the results.

### 3.9.2 Speculative Parallelization of the sequential application

In order to speculatively parallelize a sequential code there exist some questions that have to be answered:

- *¿What lines should be parallelized?* First of all, it is necessary to know what are the lines of the code that can be parallelized. In this example is a simple step because there is a single loop. In the case of there exist several loops, all of them could be parallelized if they are not nested. Also, it is recommended that loop to be parallelized has a high number of iterations in order to extract a high performance. Good performance is not the objective of the example shown, it is to show how could an application be parallelized.

  In the case of nested loops only one of the loops can be parallelized: the outermost, the innermost, or any of the intermediate loops.

- *Which are the speculative variables?* With this question we refer to those shared variables of the loop that can induce dependence violations during parallel execution of the loop. In this case, four variables are use inside the loop: `P`,`Q`,`aux` and `vector`. The first three variables modify their values at the beginning of the loop, before they were used. Therefore, they do not induce dependences and can be considered private variables. On the other hand, the fourth variable is shared by all iterations of the loop. If while iteration `i` was writing position `k` of the vector, the iteration `i-1` was reading the same position `k`, a dependence violation arises. So, the speculative variable is `vector`.

- *Which is the size of the speculative variable?* Once known which is the speculative variable, its size should be known. To do so a file called `myexamplevariables.h` has been implemented.

```
1  // This file includes all common variables of the example
2
3  // Constants
4  // Vector size and number of iterations
5        #define MAX 100
6        #define NITER 1000000
7
8  // Data structures
9        int vector[MAX];
```

  `vector` size is `MAX`, specifically 100.

Once answered these three questions, the modification of the sequential application can be started. First of all, we have to add some files in the application path:

- *speccode-vXX.c.* It contains the speculative engine code in its `XX` version.

- *variables.h.* This file has some configuration parameters that depends on the application to parallelize. Size of the loop, and size of the speculative variables are examples of data contained by this file. So, the following modifications should be performed to this file:

```
12 // number of threads to be used
13 #define threads 2
14 // window size
15 #define wsize threads*2
16 //blocking factor
17 #define blk 1000
18
19 // Application-dependant settings follow
20
21 #define shared_size 100
22 #define cur_upper_limit 1000000
23 #define max_upper_limit 1000000
```

  `threads` variable indicates the number of threads available. The *sliding window* slots are indicated by `wsize`. If the block size of iterations is statically assigned, `blk` will indicate the number of iterations of each chunk. The size of the speculative variables is indicated by `shared_size` variable. In our example, we work with a single variable (`vector`) with a size of 100. The total number of iterations to execute is indicated by `cur_upper_limit`, in this case, this value is similar to `NITER`. Finally, `max_upper_limit` has the same value as `cur_upper_limit`.

Once configured parameters of the engine, original code is modified. First the following two headers should be added to the example file `example-vBRA09.c`:

```
9  //   speccode: OpenMP header file included
10       #include <omp.h>
```

```
13  //   speccode: Main header file with all common variables
14       #include "specEngine.h"
```

The next step is to add another line to initialize the engine structures:

```
27  //   speccode_ Initializing structures
28       specinit();
```

Before the beginning of the loop, the following lines are inserted. They are related with the use of OpenMP (except `specstart(NITER)` function that belongs to speculative library):

```
60  //    speccode: OMP threading directive
61        omp_set_num_threads(threads);
62  //    speccode: initializing speculation structures
63        specstart(NITER);
```

```
70  // Beginning of the parallel part
71  //     speccode: Speculative loop
```

```
74  #pragma omp parallel default(none)                          \
75              private(aux, Q, P, value, linear,                 \
76                      current, tid, retflag, my_IV_tail, nonSpeculative)   \
77              shared(vector, wheel_ns, wheel_ms, shadow,     \
78                      AV, IV, IV_tail, gELV, wheel, wheel_ol,          \
79                      upper_limit, varblock, jmpbuf, endLoop, endReturn)
80  {
81
82  #pragma omp for \
83     schedule(static)
```

Line 61 indicates the number of threads to be used in parallel. Lone 63 initializes speculative engine structures.

Line 74 is an OpenMP directive. It is used to mark the start of the `for` loop to be parallelized. From line 74 to 79 clauses of directive `parallel for` are found. In the line 74, `default(none)` indicates to OpenMP libary that classification of variables will be performed manually. Most of variables classified are used by the engine, with the exception of `P, Q, aux` and `vector`, but this variables have been already explained: The first three are private variables, while the last one is shared. Finally, line 83 indicates that the size of the chunks of iterations used will be static.

The following instruction:

```
47  for (P=1 ; P<=NITER ; P++)
```

is replaced by:

```
84  // for (P=1 ; P<=NITER ; P++)
85  initLoopSpecEngine(vector,P,1,1);
```

In this context, a loop will be executed in parallel with `P` as index, from value 1. The last parameter is used to identify the loop.

The following step is search all the lines where speculative variable appear. In this way, `vector` is used in lines 53 and 57 of sequential code. The first one is a load operation, therefore, is replaced by *specload* function:

```
75  //*************************************************************************
76  // speccode: speculative load. Original line:
77  //    aux = vector[Q-1];
78        linear = (Q-1);
79        if( specload(linear, current, &value, (int *) vector, &my_IV_tail) == -1)
80             earlySquash(1);
81        aux = value;
82  //*************************************************************************
```

Previous section contains the description of the arguments of the function. Specifically, speculative structure `vector` and index `Q` should be indicated in order to show which item is going to be used. To do so, `linear` variable is used (previously assigned with `Q-1`). Also, the *specload* function checks if a situation of *early squashing* appears, in which case, execution of this chunk of iterations is retried.

Second use of the `vector` variable is to be written. So, it should be replaced by the function **specstore**:

```
86  //*************************************************************************
87  // speccode: speculative store. Original line:
88  //   vector[Q-1] = aux;
89        linear = (Q-1);
90        value = aux;
91        specstore(linear, current, value, &my_IV_tail);
92  //*************************************************************************
```

Previous section contains the description of the arguments of the function. Specifically, speculative structure `vector` and index `Q` should be indicated in order to show which item is going to be used. To do so, `linear` variable is used (previously assigned with `Q-1`). Also, the value to be stored is passed, in this case `aux`. Following the same direction than `linear` variable, and to ease the implementation, this variable is replaced by the auxiliary variable `value`.

Once replaced all load and store operations by their corresponding library functions, an additional line should be added in the end of the loop:

```
109   endLoopSpecEngine(vector,P,NITER,1,1);
```

In this point, application has been parallelized with the use of the speculative library described. Now, we only have to compile the code and execute it.

### 3.9.3 Resume

Steps performed to speculatively parallelize an application can be resumed in:

1. Identify the loop to be parallelized and its number of iterations.
2. Identify speculative variables and their size.
3. Configure the file `variables.h` with the number of iterations and the size of speculative variables.

4. Add the headers of OpenMP and of the speculative library.
5. Initialize the structures of the engine and state private and shared variables of the loop with the use of the OpenMP directives.
6. Introduce an initial sentence at the beginning of the speculative loop.
7. Replace load and store operations of the speculative variables with *specload* and *specstore* functions respectively.
8. Introduce a final sentence in the end of the speculative loop.

# Chapter 4

# New speculative library

*En este capítulo se describe un motor de paralelización especulativa desarrollado por el autor como Trabajo de Fin de Grado [23], y que permite solventar las limitaciones del motor descrito en el capítulo anterior.*

*Este motor también basa su ejecución en una ventana deslizante, y utiliza elementos software para especular sobre códigos secuenciales. Al igual que la librería de Cintra y Llanos [10] ya descrita, este motor deja en manos de OpenMP la resolución de los detalles internos relativos al paralelismo, por tanto, se deben clasificar las variables antes de la ejecución del bucle.*

*También se describen las nuevas estructuras de datos que se utilizarán, principalmente, una matriz para cada slot de la ventana deslizante, donde se almacenarán las variables sobre las que se especule: su dirección, tamaño, su estado, etc.*

*Además se dará una descripción de la nueva implementación de las operaciones de lectura, escritura y consolidación especulativas, adaptadas al nuevo tipo de estructuras de datos con los que tendrán que trabajar.*

*En el capítulo siguiente se expondrán las limitaciones de este motor en términos de rendimiento y su solución, que son el objeto de este trabajo.*

## 4.1 Introduction

We have developed a new TLS runtime library that supports the speculative execution of `for` loops. The library architecture follows the same design principles of the speculative parallelization library developed by Cintra and Llanos [10, 11]. In order to understand our solution, a brief description of that proposal is needed.

Cintra and Llanos [10, 11] 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:

- Their library requires that all speculative variables were packed in a single, one-dimensional vector before the start of the speculative loop. At this way, we should modify the original source code of the applications introducing some lines to define a new structure that save all the speculative variables. Moreover, this library can be used only with vectors or matrix, do not support more complex structures.
- All speculative variables should share a single data type. In fact, sharing a single, one-dimensional vector to save all the variables implies that all of them should be from a single data type: *char*, *int*, *double*, etc.
- Speculative variables can only be accessed by name inside the loop (no references by addresses or pointers were allowed).
- 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 recently accessed.

Our new thread-level speculative runtime library removes all these limitations. It allows to speculatively access variables of any data type, both by name or by address, and managing the space needed for version copies on demand. In this chapter we will briefly show the general architecture of the library.

## 4.2 Data structures

The data structures needed by the new speculative library are depicted in Figure 4.1. The sliding window mechanism is implemented by a matrix with $W$ window slots (four in the figure). Each slot acts as a "blackboard" used to handle 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 the moment. The `STATE` field indicates the state of the execution being carried out in each slot.

The figure represents the parallel execution of a loop. The loop has been divided into three chunks of iterations, and it will executed in parallel using three threads. It is

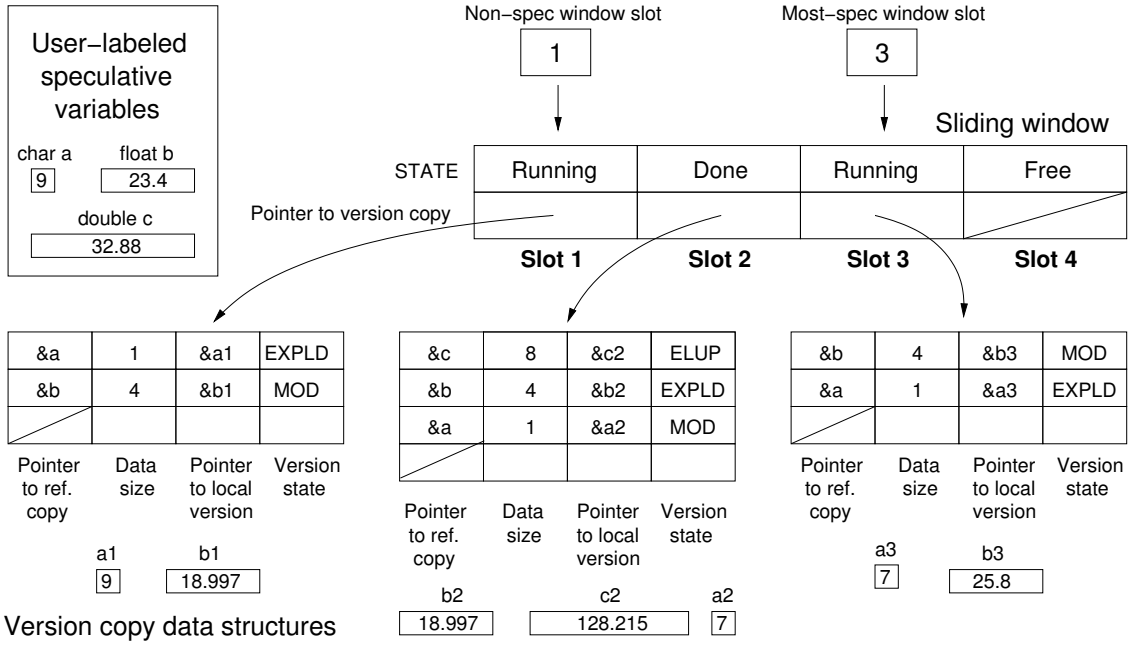



Figure 4.1: Data structures of our new speculative library.

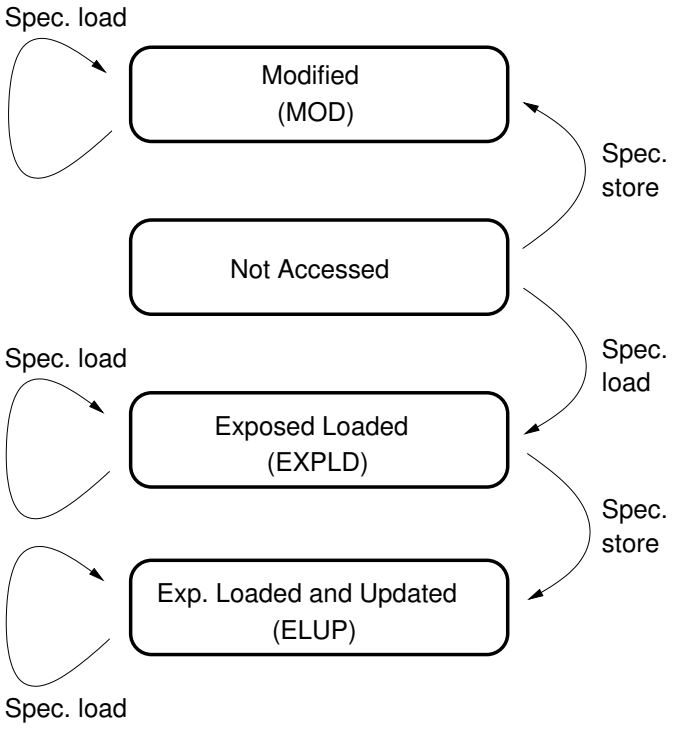



Figure 4.2: State transition diagram for speculative data.

very important to understand that there is not a 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 to maintain an order relationship among the chunks being executed.

In our example, thread working in slot 1 is executing the non-speculative chunk of iterations (as indicated by its `RUNNING` state); the following chunk has been already executed and its data has been left there to be committed after the non-spec chunk finishes (since it is in `DONE` state), while the last one, the most-speculative chunk launched so far, is also `RUNNING`. In other words, the thread in charge of the second chunk has already finished, while the non-spec and most-spec threads are working. If more chunks were pending, the freed thread would be assigned the following chunk, starting its execution in slot 4. Slot 2 can not be re-used yet, because the execution of chunk 2 left changes to speculative variables that are yet to be committed. As we will see in Section 4.5, 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 4.1 represents a situation where the programmer used three speculative variables. At a given moment, the thread executing the non-speculative chunk has speculatively accessed variables `a` and `b`. Each row of the version copy data structure keeps the information needed to manage the access to a different `speculative` variable. The first column indicates the address of the original variable, known as the *reference copy*. The second one indicates the data size. The third one indicates the address of the local copy of this variable associated to this window slot. Finally, the fourth column indicates the state associated to this local copy. Once accessed by a thread, the version copies of the speculative data can be in three different states: *Exposed Loaded*, indicating that the thread has forwarded its value from a predecessor or from the main copy; *Modified*, indicating that the thread has written to that variable without having consumed its original value; and *Exposed Loaded and Updated*, where a thread has first forwarded the value for a variable and has later modified it. The transition diagram for these states is shown in Figure 4.2.

Figure 4.1 represents a situation where the thread working in slot 1 has performed a speculative load from variable `a` (obtaining its value from the reference copy) and a speculative store to variable `b`. Regarding `a`, the figure shows that thread working in slots 3 has forwarded its value. With respect to variable `b`, the information in the figure shows that `b` was overwritten both by threads working in slots 1 and 3.

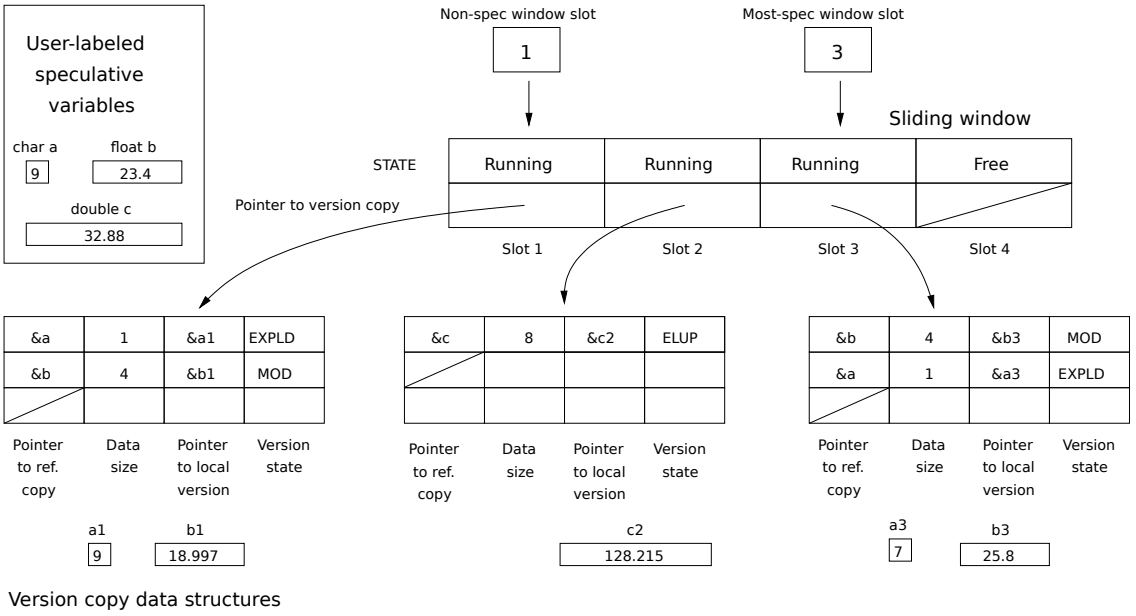
## 4.3 Speculative load

The interface of `specload()` is as follows:

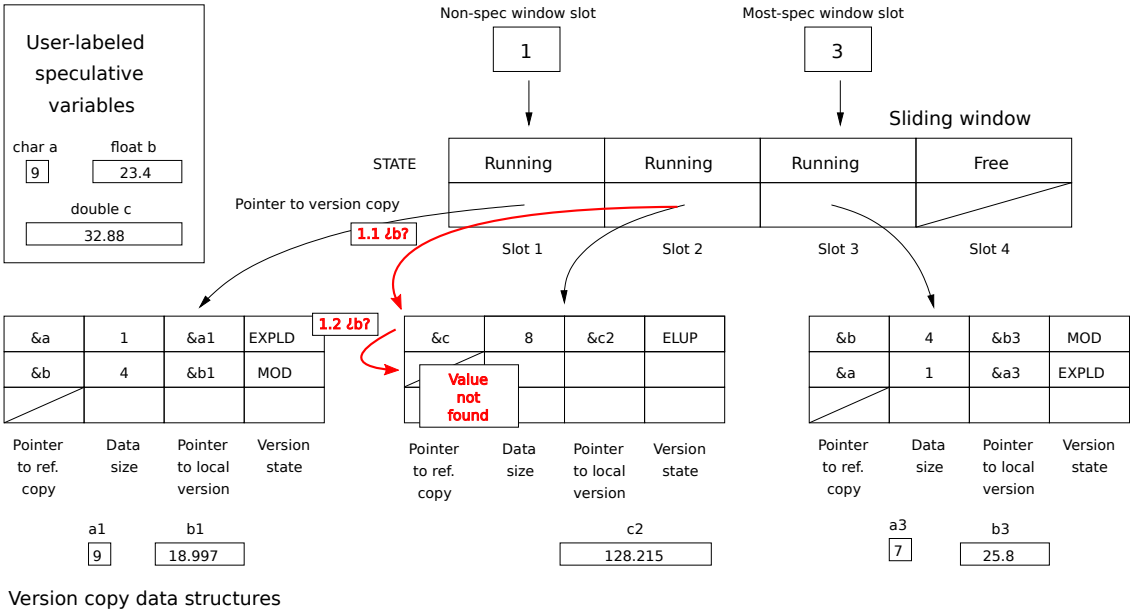
```
specload(UCHAR* addr, UINT size, UINT chunk_number, UCHAR* value)
```

Arguments have the following meaning:

1. `addr`: Is the address of the speculative variable.

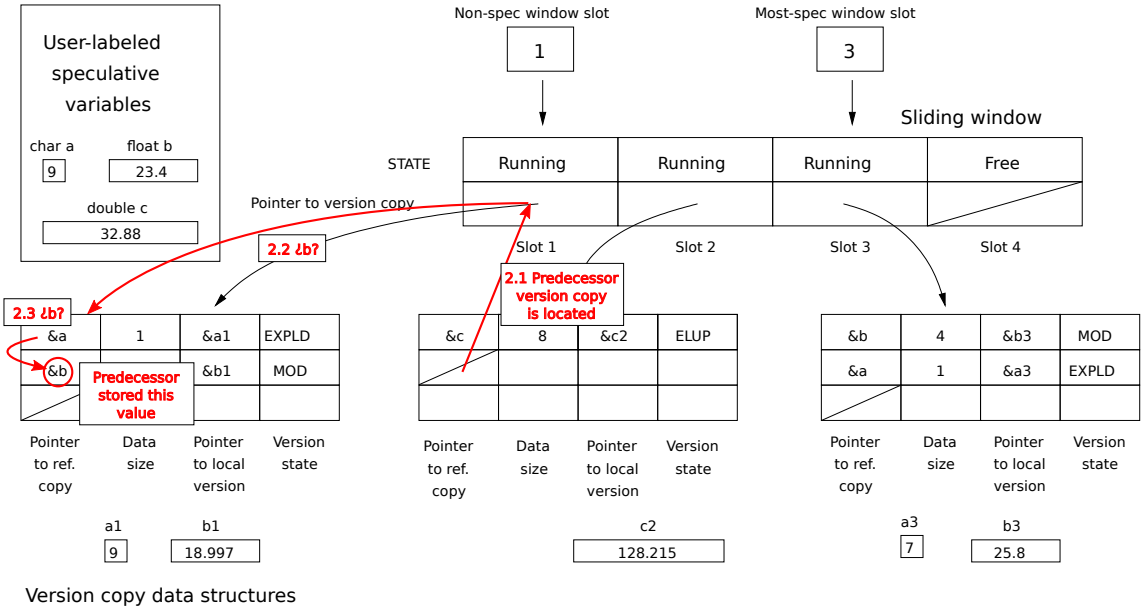



(a)

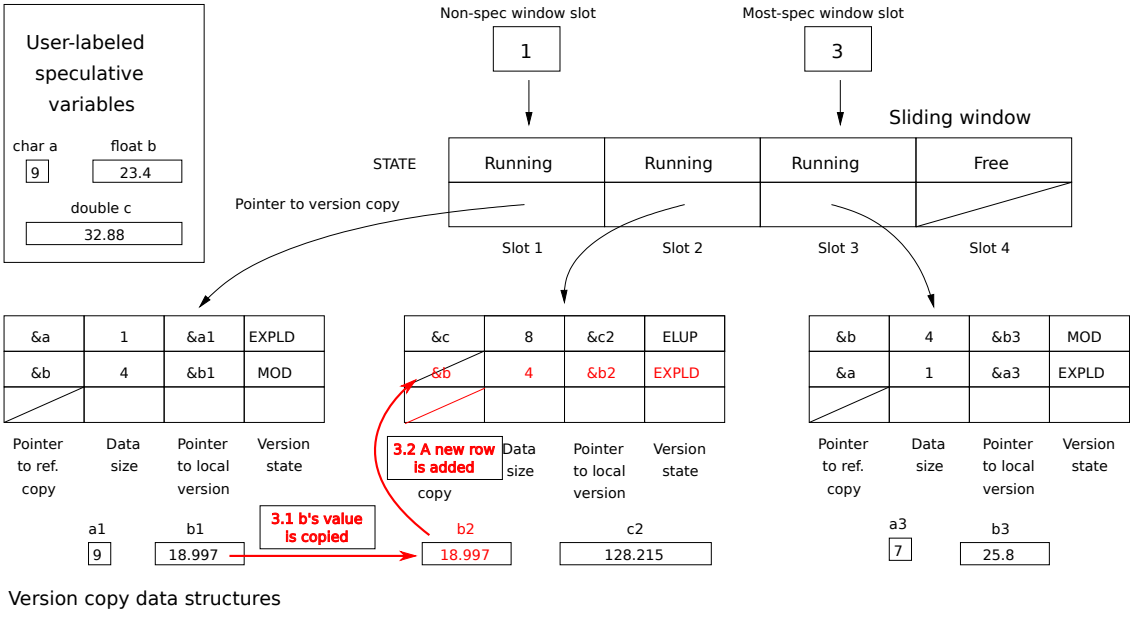



(b)

Figure 4.3: Speculative load example (1/2). (a) Initial values of the example. (b) Thread working in slot 2 scans its version copy to find the value.




(c)




(d)

Figure 4.4: Speculative load example (2/2). (c) Thread working in slot 2 goes to its predecessor version copy data structure and scans it in order to find a value for b. (d) After storing a copy of b's value, thread working in slot 2 adds a new row to its version copy data structure.

2. `size`: Is the size of the variable.

3. `chunk_number`: Is the number of the chunk being executed (needed to infer the slot being used).

4. `value`: 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. Figures 4.3 and 4.4 show how the speculative load works. Suppose that the thread working in slot 2 has only accessed to variable `c` so far (as is described in Figure 4.3(a)), and then it calls `specload(&b, sizeof(b), 2, &value)` to obtain a value for `b`. The sequence of events is the following:

1. 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 (see Figure 4.3(b)).

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 (see Figure 4.4(c)). Note that, if no value for `b` were found there, our thread would have gone to its predecessor, until the non-speculative thread were 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, 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` (see Figure 4.4(d)).

The call to `specload()` finishes returning the value $18.997$ in the address indicated by its fourth parameter.

This operation also support *early squashing* (see Section 3.2.1).

## 4.4 Speculative store

```
specstore(UCHAR* addr, UINT size, UINT chunk_number, UCHAR* value)
```

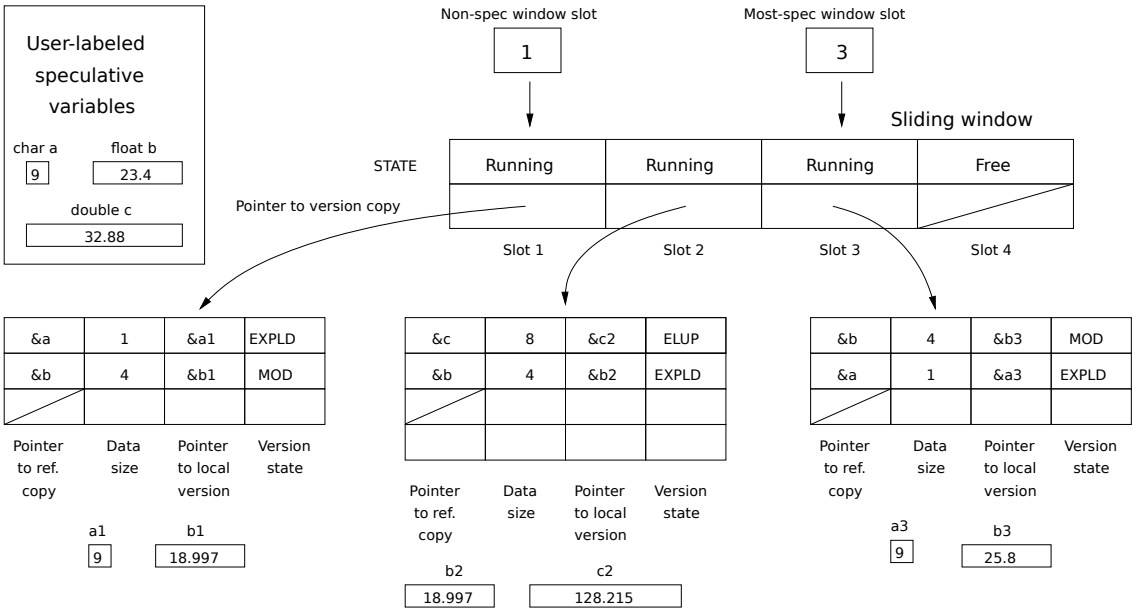
Arguments have the following meaning:

1. `addr`: Is the address of the speculative variable.

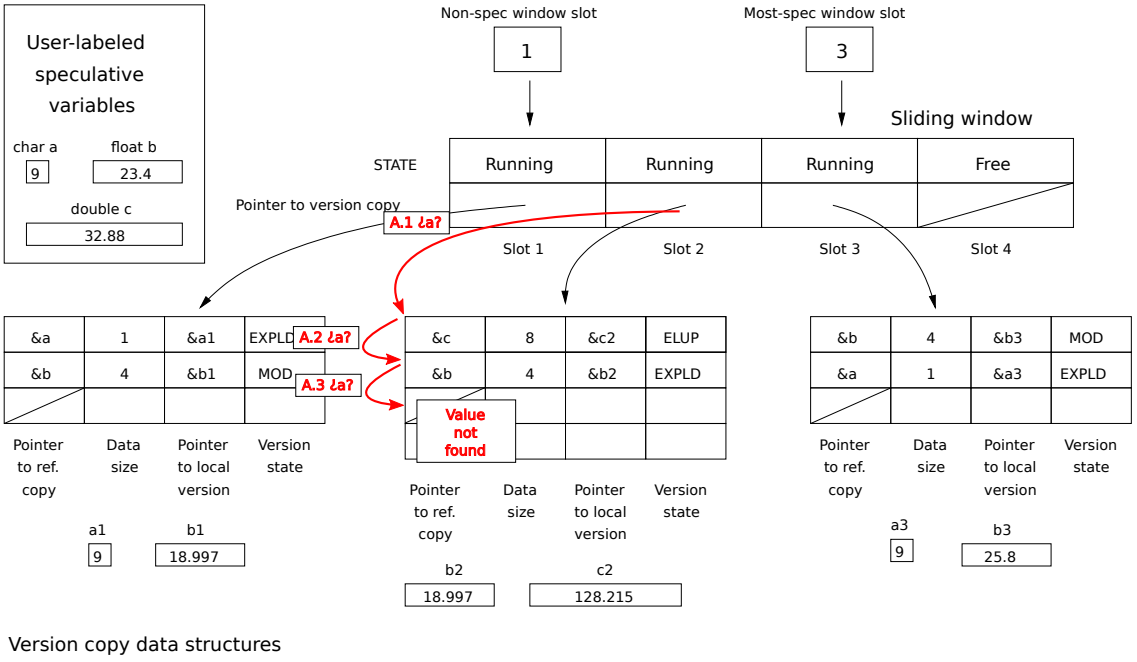
2. `size`: Is the size of the variable.

3. `chunk_number`: Is the number of the chunk being executed (needed to infer the slot being used).

4. `value`: Is a pointer to the value to be stored.

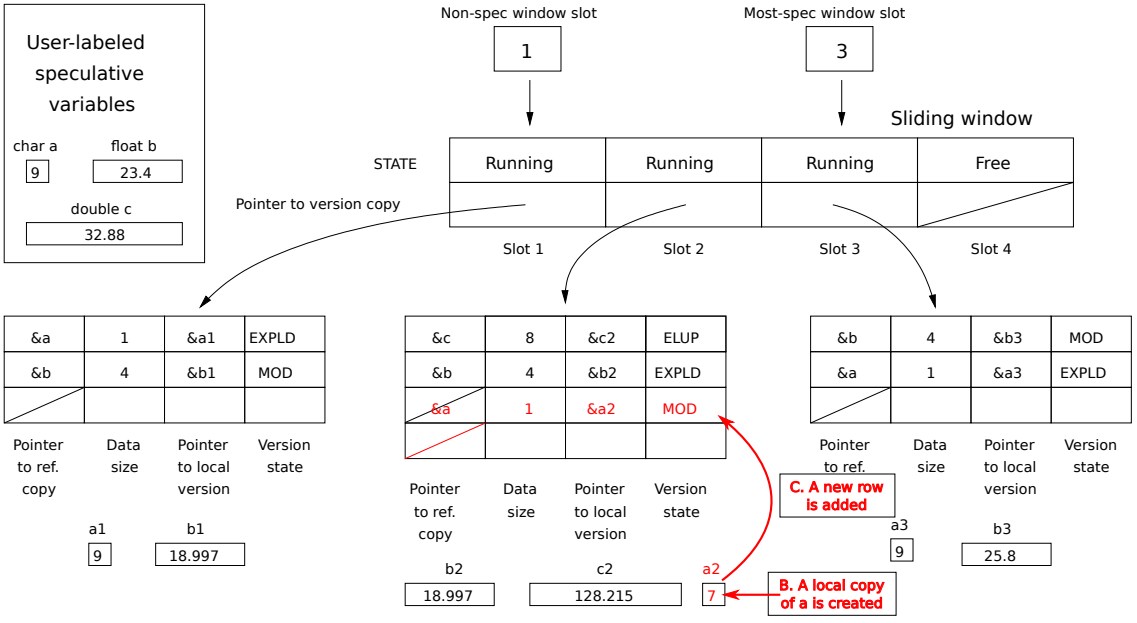



(a)



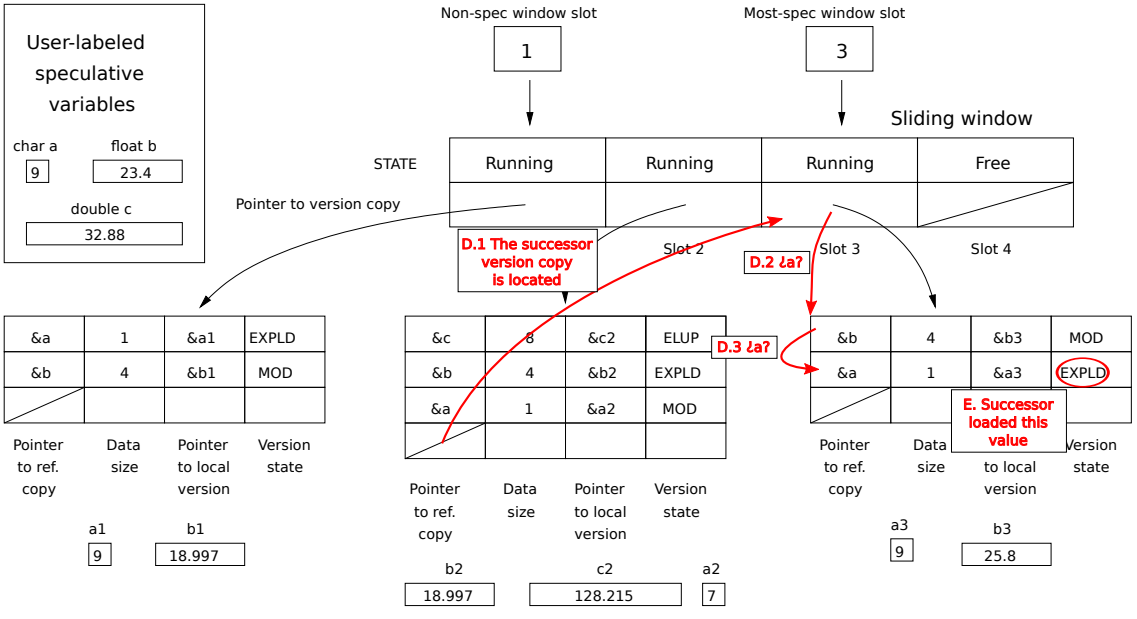

(b)

Figure 4.5: Speculative store example (1/3). (a) Initial values of the example. (b) Thread working in slot 2 scans its version copy to find the value.




Version copy data structures

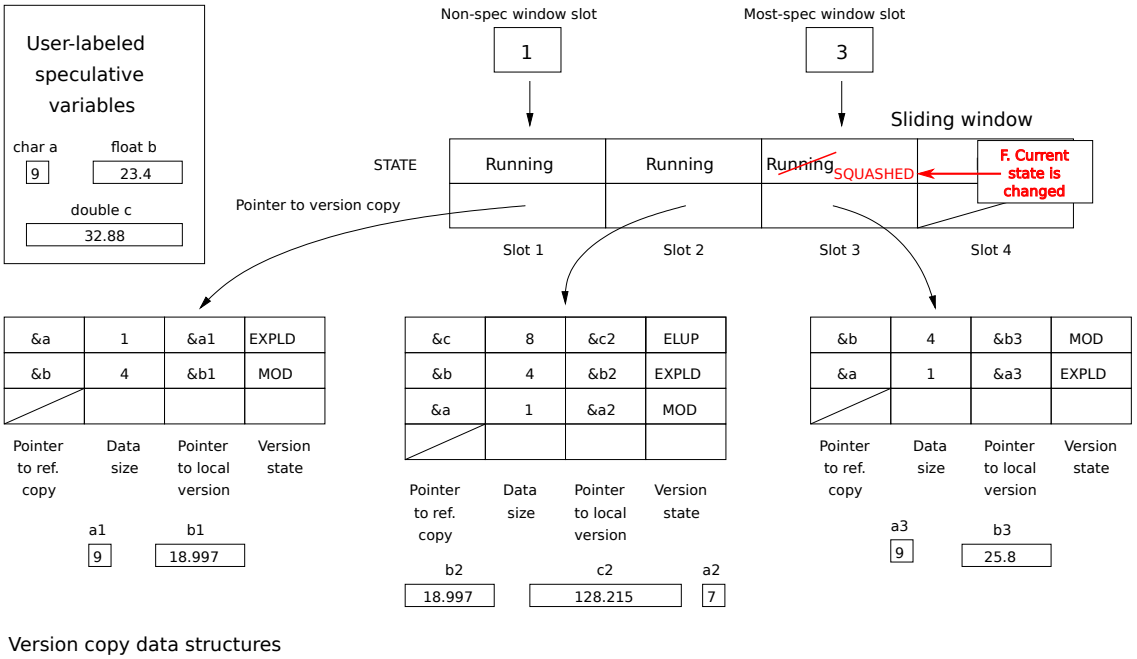
(c)



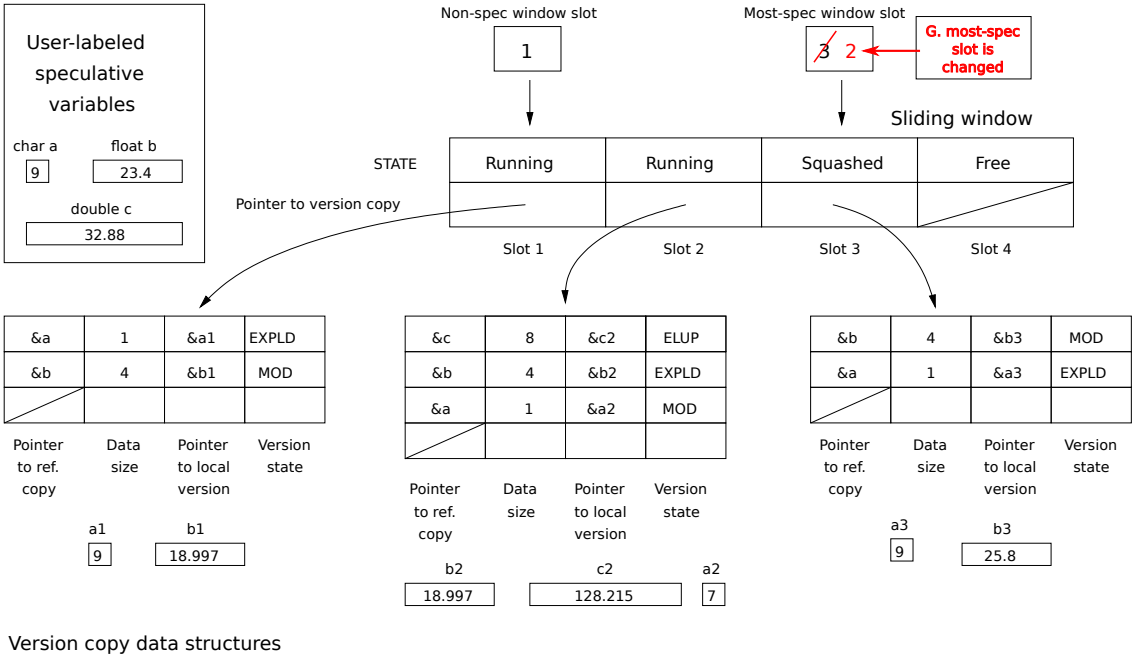

Version copy data structures

(d)

Figure 4.6: Speculative store example (2/3). (c) After creating a local copy of a, thread working in slot 2 adds a new row to its version copy data structure. (d) Thread working in slot 2 should check whether any successor has consumed an outdated value.In our example, the search finds out that thread working in Slot 3 has consumed an incorrect value for a.




(e)




(f)

Figure 4.7: Speculative store example (3/3). (e) Thread working in slot 3 should be squashed. To do so, thread working in slot 2 changes the state of slot 3 from `RUNNING` to `SQUASHED`. (f) Thread working in slot 2 marks itself as the most-speculative thread, since data stored in association with slot 3 is not longer valid.

As it can be observed, 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.

Figures 4.5, 4.6 and 4.7 show 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, taking into account that initial values are depicted in Figure 4.5(a):

A. 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 (see Figure 4.5(b)). If `a` were found, this thread would update its status according to the state diagram of Figure 4.2, and it would proceed to step D.

B. Thread working in slot 2 creates a local copy of `a`, storing value 7 on it. (see Figure 4.6(c))

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, that will be `MOD` in this case (see Figure 4.2). All these operations will be seen in Figure 4.6(c).

D. After storing the value locally, 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 `EXPLD` or `ELUP` state. These states would indicate that the successor has used the value. (see Figure 4.6(d))

E. In our example, the search finds out that thread working in Slot 3 has consumed an incorrect value for `a` (see Figure 4.6(d)). If no dependence violation was detected, the call to `specstore()` would finish here.

F. A dependence violation has been detected. Thread working in slot 3 should be squashed. To do so, thread working in slot 2 changes the state of slot 3 from `RUNNING` to `SQUASHED` (see Figure 4.7(e)). Since all threads check their own state at the beginning of each `specload()` and `specstore()` call, 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.

G. Finally, thread working in slot 2 marks itself as the most-speculative thread, since data stored in association with slot 3 is not longer valid (see Figure 4.7(f)). The most-spec pointer will be advanced later by the thread that will receive the task of re-executing chunk 3.

If, after these events, thread working in slot 2 finishes its execution, while threads associated to slot 1 and 3 are still working, we arrive to the situation shown in (see Figure 4.1). 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.

## 4.5 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 is the non-speculative. If the thread is speculative, the slot is left to be committed by the non-spec thread.

As in the case of previous operations, let us examine an example case: Suppose that we are in the situation depicted in Figure 4.8(a), and the thread working in slot 2 finishes.

I At this time, thread working in slot 2 should change its state in order to show other threads that its chunk of iterations have been executed, and should be committed(see Figure 4.8(b)). However, this could only be performed by the non-speculative thread.

II After that, the non-spec thread working in slot 1 finishes, therefore begins to commit all of its values. Operations performed to carry out this task are depicted in Figures 4.9(c) and 4.9(d). For example, `b` element should be committed, so it copies the content of `b1` into `b`.

III When no more elements are available, i.e., after committing the version copy data structure associated to slot 1, it changes its state to `FREE`. (see Figure 4.10(e)).

IV Then this thread checks if any successor thread has finished its execution. In our example the thread working in slot 2 has finished, so its elements must be committed. (see Figure 4.10(f)).

V Elements of the slot 2 are committed following the order depicted in Figures 4.11(g), 4.11(h) and 4.12(i).

VI When no more elements are available, the state associated to thread working in slot 2 is changed to `FREE`. (see Figure 4.12(j)).

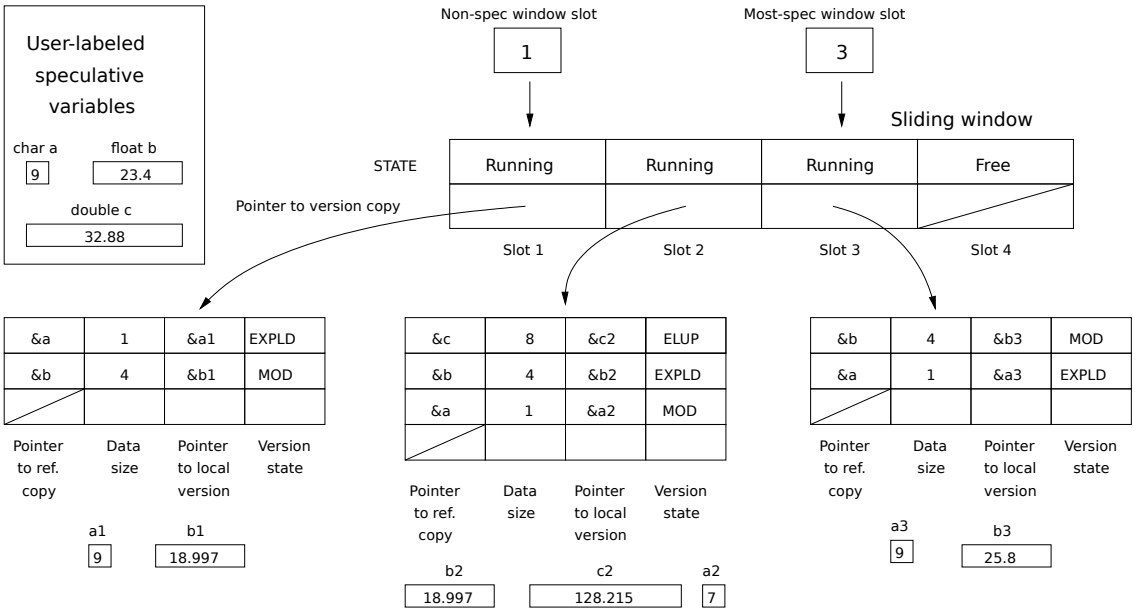
VII The state of the thread working in the next slot is not `DONE`, then, commit operation could be finished. Nevertheless, non-spec pointer should be advanced to this slot 3. (see Figure 4.13(k)).

After these operations the situation depicted in Figure 4.13(l) is achieved. In this way, version copies of slots committed are not entirely reset until another chunk of iterations is assigned to them, and change its state to `RUNNING`.

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

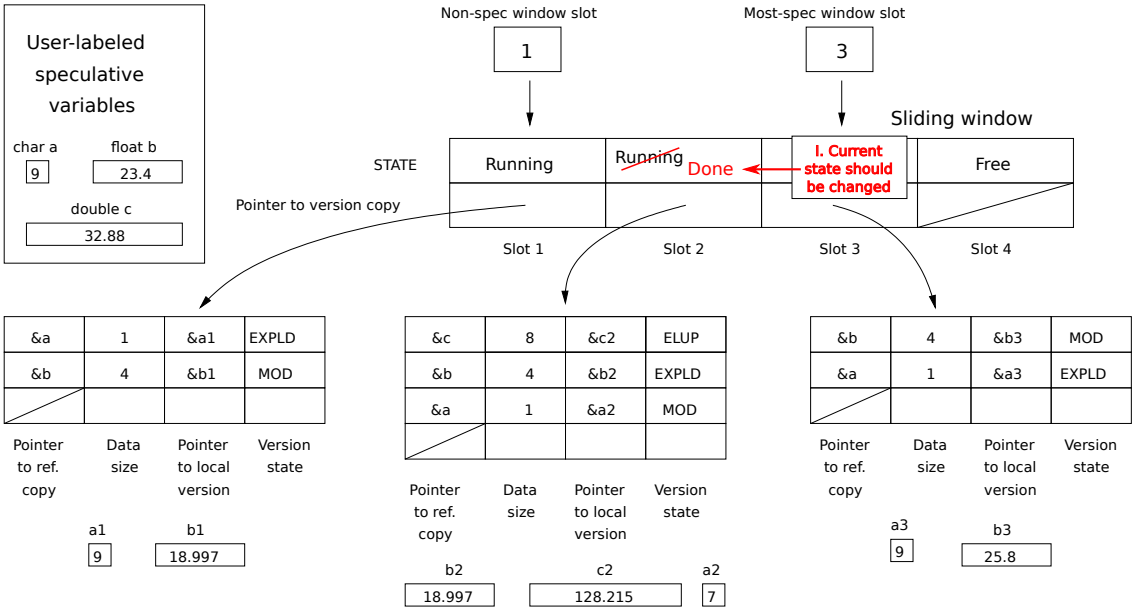
## 4.6 Initialization functions of the engine

This speculative library uses some data structures that should be initialized before the execution of any speculative code. In this way, the two functions introduced in Section 3.7 have been re-used, but now they have a little bit different behaviour:




Version copy data structures

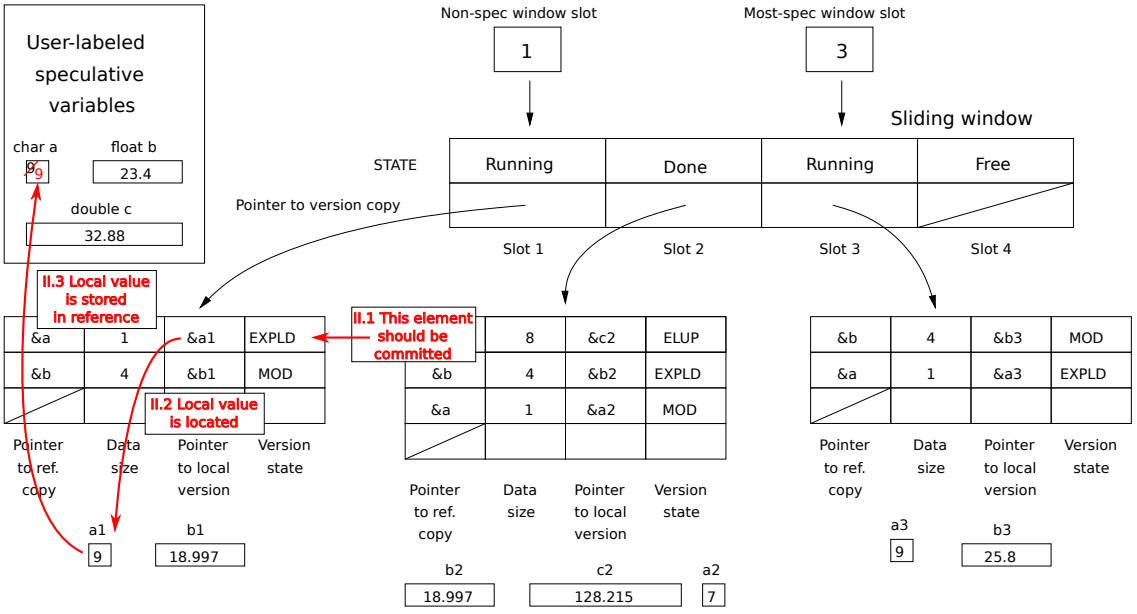
(a)




Version copy data structures

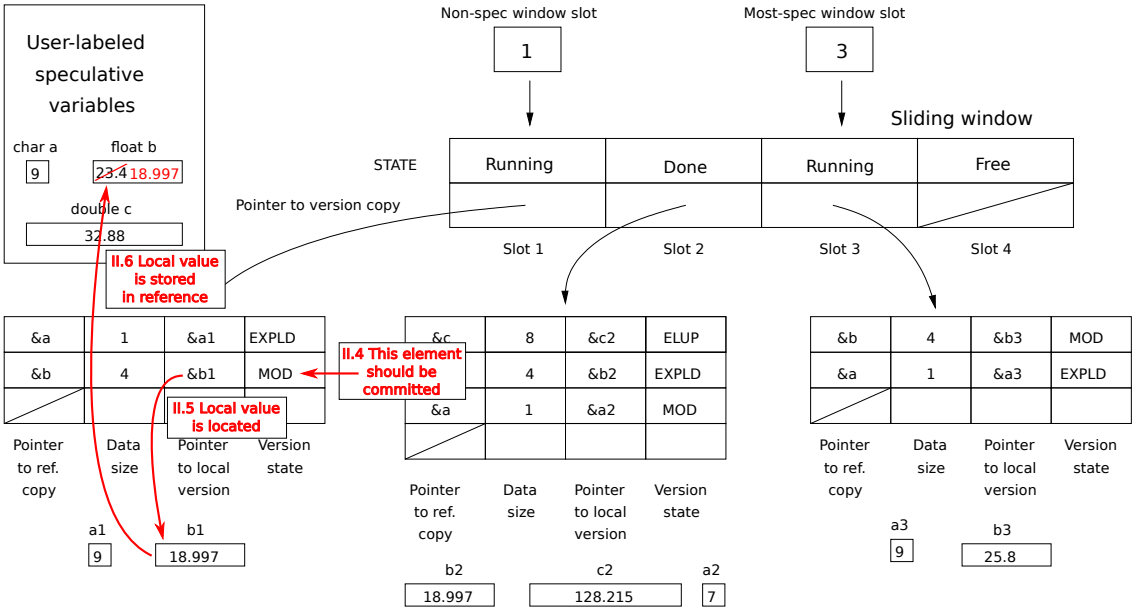
(b)

Figure 4.8: Speculative commit example (1/6). (a) Initial values of the example. (b) Thread working in slot 2 finishes its chunk of iterations.




Version copy data structures

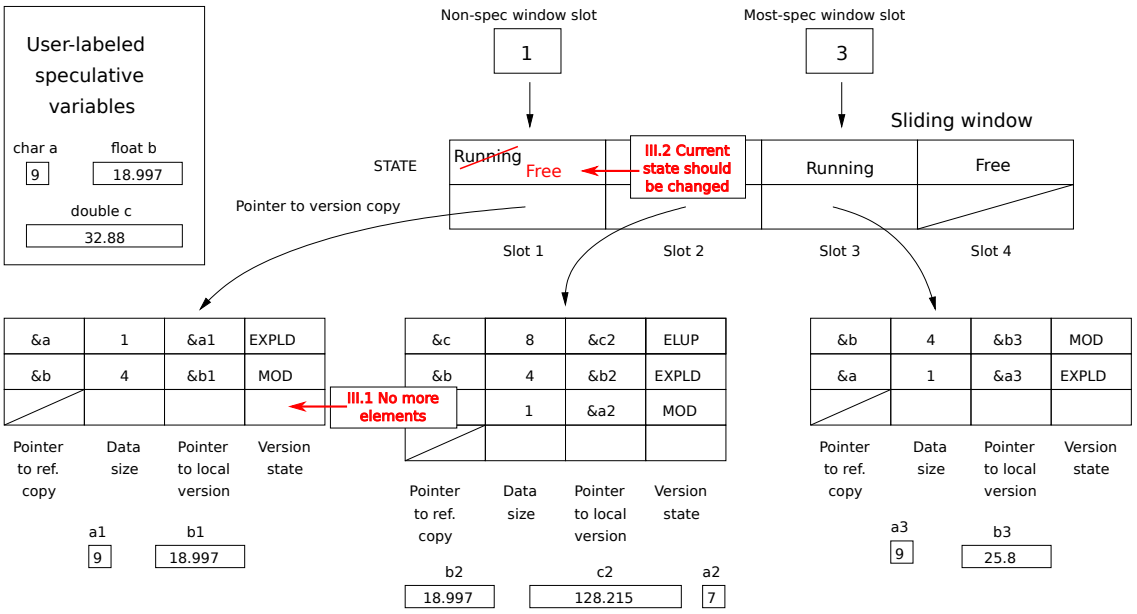
(c)




Version copy data structures

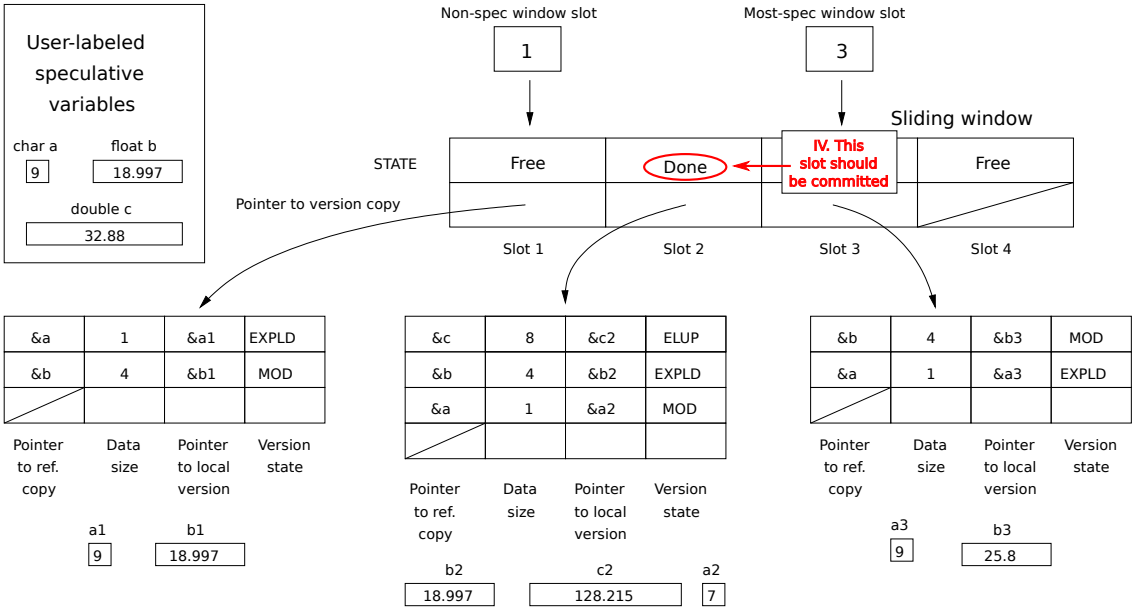
(d)

Figure 4.9: Speculative commit example (2/6). (c) Non-speculative thread finishes its execution so its elements start to be committed. (d) The next value of the non-speculative thread is committed.




Version copy data structures

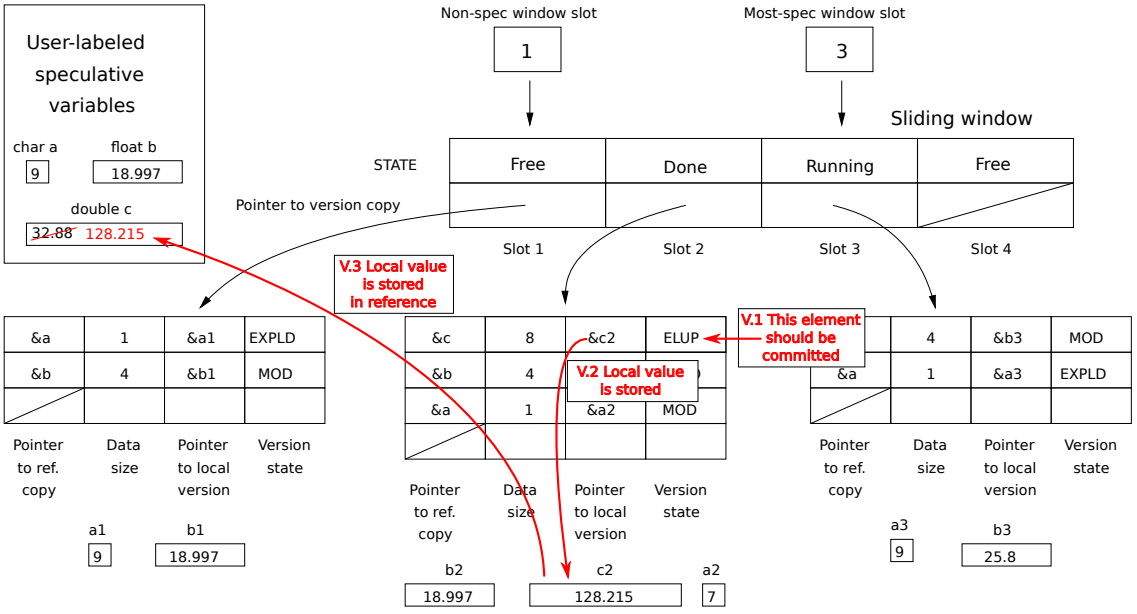
(e)




Version copy data structures

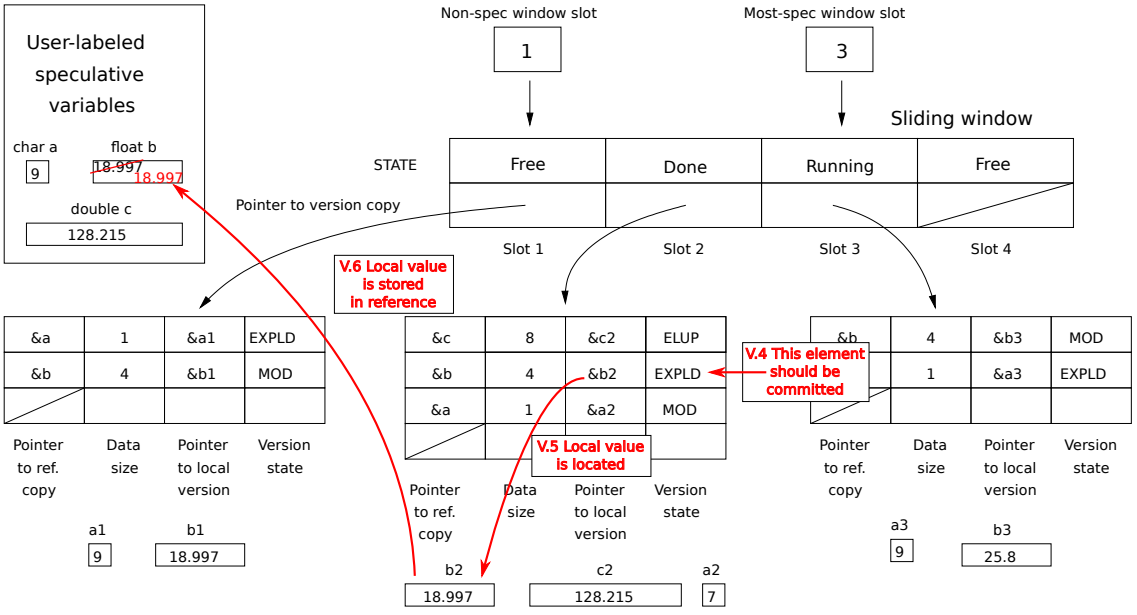
(f)

Figure 4.10: Speculative commit example (3/6). (e) No more elements are located in the version copy of the non-speculative slot, so thread working in slot 1 changes its state from `RUNNING` to `FREE`. (f) After change its own state, thread working in slot non-speculative, checks if successor slots could be committed, i.e., if their states are `DONE`.




Version copy data structures

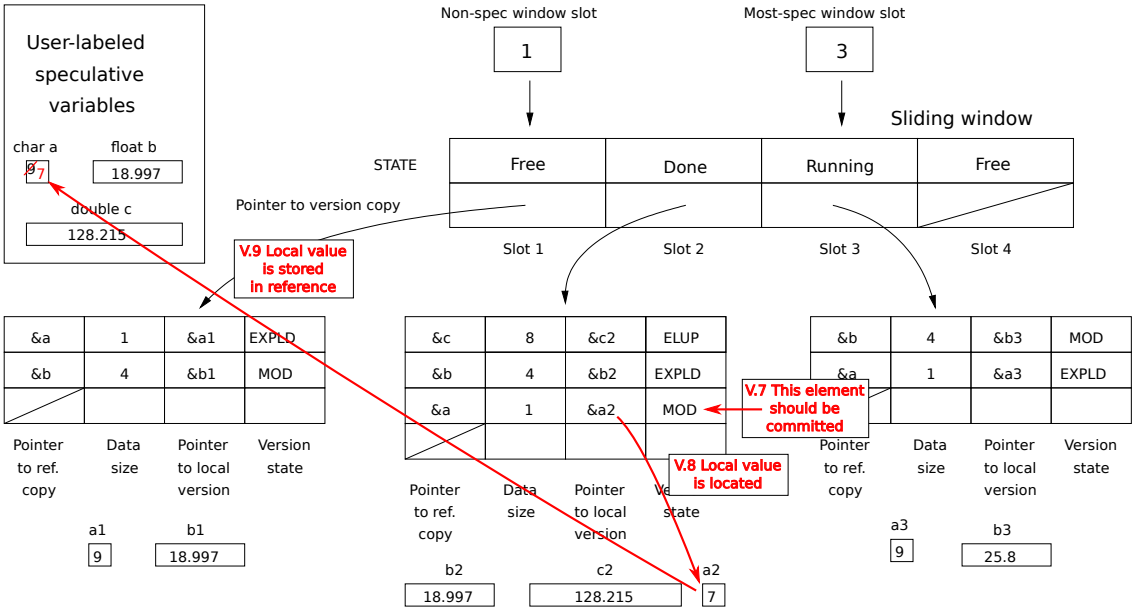
(g)




Version copy data structures

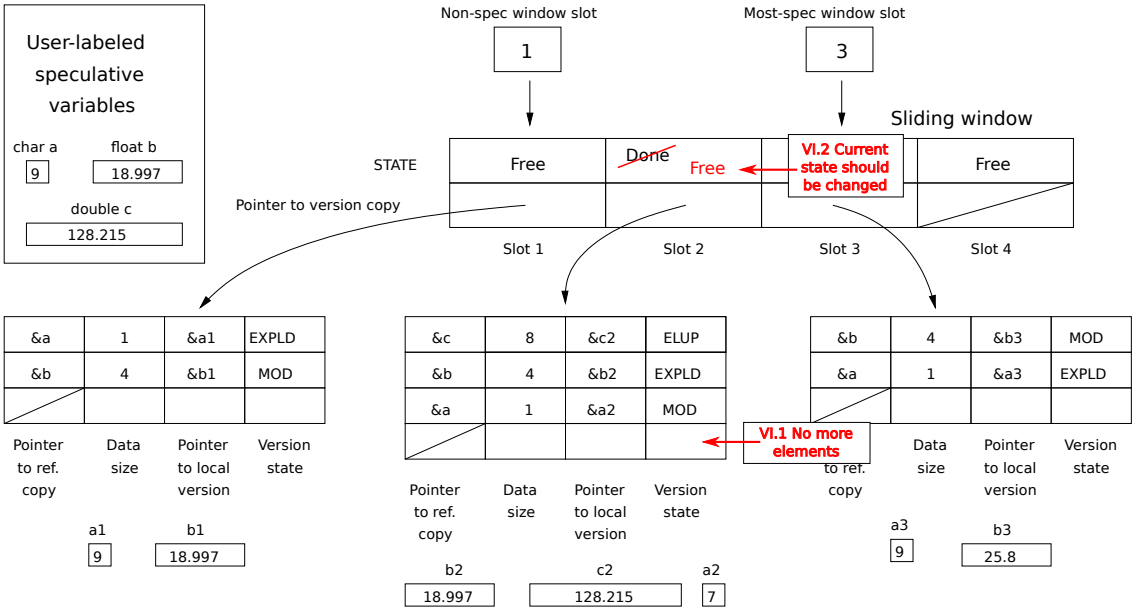
(h)

Figure 4.11: Speculative commit example (4/6). (g) Thread working in slot non-speculative starts to commit elements from slot 2 because its state was `DONE`. (h) The next element from slot 2 is committed.




Version copy data structures

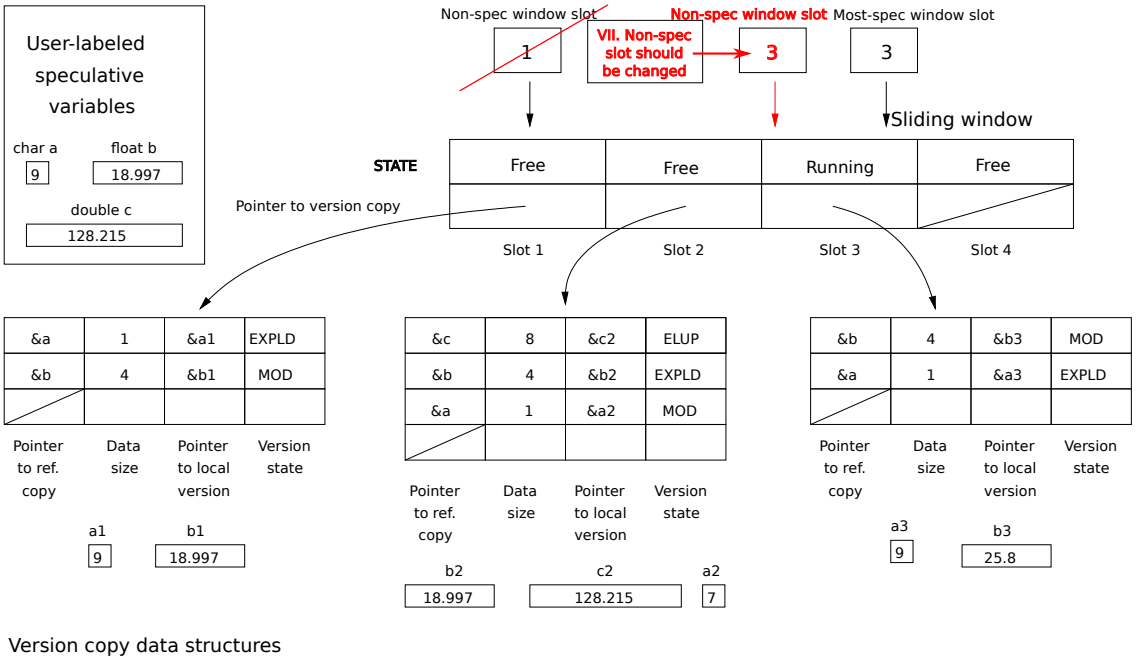
(i)



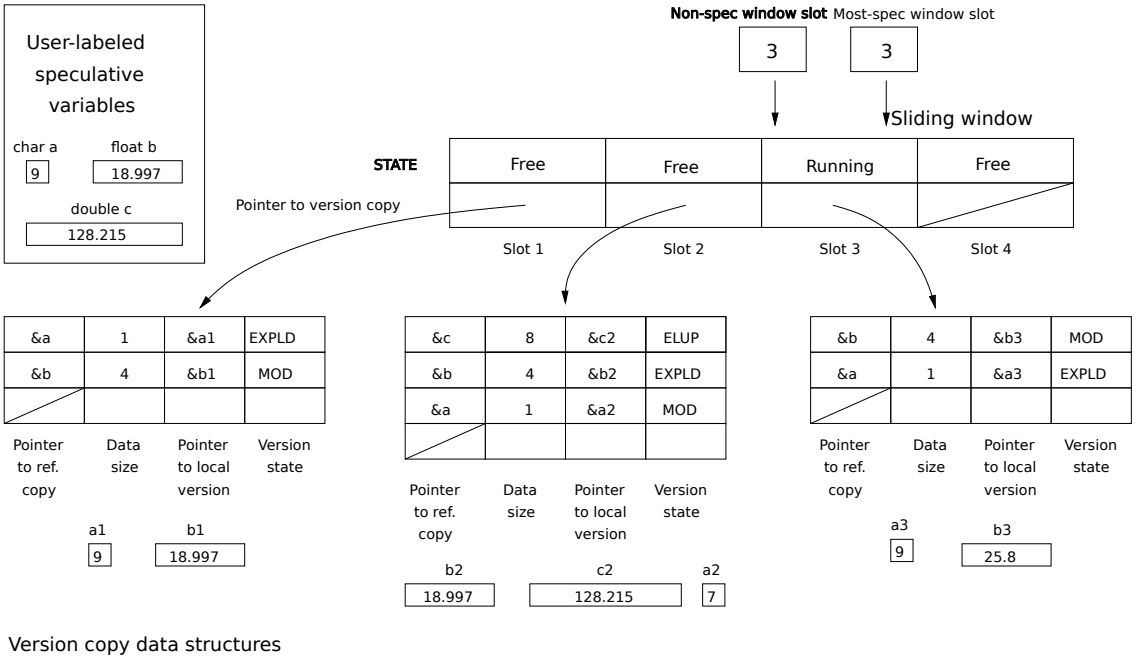

Version copy data structures

(j)

Figure 4.12: Speculative commit example (5/6). (i) The last element from slot 2 is committed. (j) There are not more elements to commit, so, the state of this slot is changed from DONE to FREE.




(k)




(l)

Figure 4.13: Speculative commit example (6/6). (k) No more slots could be committed, therefore, pointer to non-speculative slot is updated. (l) Final situation.

- *specinit*: This function should assign the static, or dynamic, block size that slots will use, and allocates memory needed by dynamic data structures used by this library.

- *specstart*: This function should initialize non-speculative and most-speculative pointers, states of the sliding window to the FREE state, and assigns the limit value of iterations at run-time with the use of the single integer argument that use this function: *specstart(int iterations)*.

## 4.7 Use of the engine and variable settings

In a similar way than the previous engine, this library should implements some variables:

- *threads*: Indicate available threads to execute the problem.

- *wsize*: Window size, that is, the number of slots.

- *blk*: When block size is static, this variable implements this size, specifically, the number of iterations that form blocks.

- *maxiter*: Maximum number of iterations of the loop to be parallelized.

Once implemented the values of these constants, some changes should be applied in the original code: We should include OpenMP library, call initialization functions (*specinit* and *specstart(int iterations)*), set the number of threads to be used (with the function *omp_set_num_threads(int threads)*) and classify variables into two types: *private* or *shared*; this is one of the requirements of OpenMP[1, 8, 15].

Standard variables to be classified are less than in the previous version (see Section 3.8):

- *Private*: current, tid, retflag.

- *Shared*: wheel_ns, wheel_ms, wheel, upper_limit, varblock.

Speculative parallelization of the loop could be implemented using OpenMP directives. To perform load, store, and reduction operations over speculative variables should be replaced original instructions with the functions described along this chapter.

Finally, when the execution of a thread ends, partial commit operation is performed with the use of the *threadend* function, and this thread checks if there are any more blocks to be executed.

## 4.8 An example of use of this library

An example developed in C language is going to be shown in order to know how to use the described engine to speculative parallelize an application. The application of the example will be the same of the described in the previous chapter, in Section 3.9. So in this case the sequential code is omitted (see Section 3.9.1).

### 4.8.1 Speculative Parallelization of the example

This new version of the library has only a few interface differences with the previous one. So, only the main operations will be detailed, specifically, *specload* and *specstore* operations. However a briefly resume of the first steps is described below (For more information see Section 3.9.2):

1. Additional, headers and internal variables are defined.
2. A call to `specinit()` initializes the runtime speculative library. This is required only once for the entire application.
3. Before the loop, the `omp_get_num_threads()` function is called to obtain the number of available threads.
4. A `specstart()` function is called to initialize the execution of the following parallel loop.
5. All variables labeled are labeled as `private` or `shared`.
6. The original loop structure is replaced with a parallel `for` loop with just "`threads`" iterations. This launches the number of desired threads.
7. A `while(true)` loop ensures that each thread repeatedly requires a chunk of iterations from the original loop to be processed. If no chunks are left, a `break` statement exits this loop and the end of the thread is reached.
8. Inside the loop, each thread receives the index of the first iteration of its assigned chunk and proceeds with the original loop body.

Now, all the lines where speculative variable appear should be modified. The first load operation (see line 53 of sequential application in Section 3.9.1) is replaced by `specload` function:

```
//*********************************************************************
// speccode: speculative load. Original line:
//    aux = vector[Q-1];
//    linear = (Q-1);
//    if( specload(linear, current, &value, (int *) vector, &my_IV_tail) == -1)
    if (specload((unsigned char *) &(vector[Q-1]), sizeof (vector[Q-1]), current,
         (unsigned char *) &value) == -1)
         earlySquash(1);
    aux = value;
//*********************************************************************
```

The code shown above contains old, and new, `specload` calls, in order to have a better understanding of the differences between the two functions. For example, now, `linear` variable is not used because the address of the variable is directly passed. This function recovers the most up-to-date value for this variable. The exact behavior of `specload()` has been described in Section 4.3. The value is stored in a private, temporal location.

The second operation related to speculative variables is a store in the line 57 of the sequential application (see Section 3.9.1):

```
//*********************************************************************
// speccode: speculative store. Original line:
//    vector[Q-1] = aux;
//    linear = (Q-1);
    value = aux;
//    specstore(linear, current, value, &my_IV_tail);
    specstore((unsigned char *) &(vector[Q-1]), sizeof (vector[Q-1]), current, (
        unsigned char *) &value);
//*********************************************************************
```

In the same way than the previous load operation, two versions of the function have been exposed. The new `specstore` function stores the value in a local version copy and then checks whether a successor has already consumed an outdated value of `vector[Q-1]`. If so, the offending thread and some or all of its successors (depending on the squash policy being defined [30]) are squashed. This function has been described in depth in Section 4.4.

Finally, let us resume the remaining changes that should be applied to the original code:

10. Once finished the original loop body, a call to `commit_or_discard_data()` checks whether the thread has been squashed or not. If a squash operation was issued by a predecessor, local copies of speculative data will be discarded. If the thread has not been squashed and it is the not-spec one, a partial commit will occur. Partial commits have been described in Section 4.5.

11. 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.

# Chapter 5

# Performance limitations and proposed solutions

*La librería de paralelización especulativa descrita en el capítulo anterior soporta la inmensa mayoría de las aplicaciones secuenciales, sin embargo, en la actualidad sus tiempos de ejecución son muy deficientes, llegando a superar los obtenidos por las aplicaciones en una ejecución secuencial, por tanto, su funcionalidad sería nula.*

*Para localizar los principales cuellos de botella de nuestro nuevo motor especulativo, por una parte, se ha medido a través de llamadas al sistema el tiempo de ejecución de las operaciones especulativas implicadas en la ejecución, obteniendo que las operaciones de lectura y escritura especulativas son los principales cuellos de botella del sistema. Por otra parte, se han repetido las medidas con la ayuda del profiler VTune Amplifier XE 2011 y el uso de ICC, obteniendo el tiempo de cada operación, pero evitando así las llamadas al sistema. En ambos casos las conclusiones fueron similares, es decir, el sistema perdía mucho tiempo en las operaciones de lectura y escritura especulativas. Se dedujo que el recorrido secuencial de las variables que utiliza cada slot para especular era el principal problema, y por tanto, se optó por aplicar tres posibles mejoras al sistema actual.*

*La primera de ellas se basa en reducir las llamadas al sistema operativo. En el sistema original, cada operación de escritura o lectura que accedía a una variable que no hubiera sido utilizada previamente, requería una primera llamada al sistema para reservar memoria, y otra posterior para liberarla. Con la ayuda de un vector auxiliar, conseguimos evitar esta reducción del rendimiento.*

*Otra mejora aplicada a la consolidación de los datos es el uso de un vector que almacene qué variables, de las utilizadas, han sido modificadas. Con esto conseguimos que sólo estas variables sean consolidadas, sin necesidad de revisar variables que sólo hayan sido leídas. Para lograr esto, se ha utilizado un vector por cada slot de la ventana deslizante.*

*Por último, la tercera de las mejoras consiste en una estructura tridimensional para almacenar los datos y conseguir reducir los tiempos de acceso a las copias locales de los datos. Para ello se sustituye la matriz de datos*

*de cada slot, donde se almacenaba la copia local, por una matriz tridimensional, donde la dimensión adicional sea una dirección hash a las variables utilizadas.*

## 5.1 Locating bottlenecks in the TLS engine

In order to find the bottlenecks that our speculative engine have, we examined the source code in detail to extract some ideas of the tasks that require more time.

The main problem we have located is related with one of the new functionalities implemented. 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 data being speculatively accessed. This design decision comes at the cost of longer times to find the most-up-to-date value in speculative loads, and longer times to detect dependence violations in speculative stores, since both operations should traverse all the values accessed by all the predecessors and successors, respectively. Being $T$ the number of threads, in [10], this operation was in $T \times O(1) = O(T)$, since all the memory needed to any data that might be accessed was allocated in advance. In our scheme, being $N$ the number of data elements stored locally, the search is done in $T \times O(N) = O(TN)$.

In order to search for some results that endorse our theories, we implemented some auxiliary structures to store time measurements of all the functions involved in the execution. We can take a view in the following code:

```
// time vectors
double time_specload[threads];
double time_specstore[threads];
double time_threadend[threads];
double time_commit[threads];
```

The vectors shown above save the time spent in the main speculative operations carried out by each thread. With their help, we can collect information about the average time that each function spent. In addition, we have included some code in the functions to perform the measures. In the following example, we can see the additional code implemented in the *specload()* function:

```
At the beginning of the function...
#ifdef TIME_SPECLOAD
  double time_ini, time_end;
  int id = omp_get_thread_num();
  time_ini = get_time();
#endif
...
At the end of the function...
#ifdef TIME_SPECLOAD
  time_end = get_time();
  time_specload[id] += (time_end - time_ini);
#endif
```

Where *get_time()* is a function that returns the current time in microseconds. As we can see, we have used an `#IFDEF` clause, in this way, we can measure a single function, or all of them in the same execution. This parameters allow us to get only the interesting values of each experiment. Experimental results show us that the main bottlenecks were in

the *specload()* and *specstore()* functions, thus reinforcing our initial guess that to traverse all the values of predecessors, or successors, respectively, was the main bottleneck.

However, we decided to perform a tighter measures because of the lack of the use of time counters, due to multitasking issues. Therefore, we perform a deeper analysis using *ICC (Intel C compiler)*. This compiler allows to profile the function's time just by modifying the compilation flags of the application. To do so, we added the flag `-profile-functions`

However, one of the disadvantages that presents this software is that the use of `OpenMP` with profiling functions is not allowed. Therefore we could use this measurement strategy only in experiments with a single thread. Nevertheless, this requirement was acceptable because it allowed us to extract a measure of the most time-consuming functions of the software.

In this way, we needed to install the *VTune Amplifier XE 2011* to be able to understand the acquired results (some XML files produced after executing the application with the mentioned compilation flag). Once installed, and executed, we could see that results were similar than those obtained with the calls to the *get_time()* function. So, it becomes clear that the best way to improve the performance of the speculative parallelization library was to reduce runtime of *specload()* and *specstore()* functions.

Some of the main ideas developed to enhance the performance of these functions are described below. However, for simplicity we will not give all the internal details, such as, instructions reordering or changes in the kind of the iterator loops.

## 5.2 Reducing operating system calls

One of the problems detected was the excessive number of calls to the *malloc()* and *free()* functions. To better understanding why, we will use an example. Suppose that a thread executes one of the main speculative functions, i.e., *specload()* or *specstore()*. In this context, thread searches in its matrix for the address of the datum being accessed. Imagine that the datum has not been used yet, so it should be added to the matrix. In this process of attaching the new data to the matrix of the thread, we have to allocate some memory to store the local copy of this datum, therefore, the *malloc()* function should be called. The reader can better follow this example with the help of Figures 4.3 and 4.4 (specload), or in Figures 4.5, 4.6 and 4.7 (specstore) depicted in Chapter 4.

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

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

This new vector is needed by all the available threads. We have to perform an initial call to the *malloc()* function to allocate the memory of the vector of each thread, in the

same way, a final *free()* call is used to free the memory allocated. In this way, we only have to call *malloc()* again if the vector is full, i.e., if all the space reserved to store data has been used, and this occurs rarely. This solution greatly improves the performance observed.

### 5.2.1 Implementation details

The new structure modifies the basic structures of the architecture: Initially we had an structure with four entries, where one of them was a pointer to the local copy of the datum. Instead, the new approximation manages an `offset` for each datum. In this way, each datum will be stored in the `Local Version Data` vector in the position pointed by its offset, from this position to the same position plus the size of the datum, i.e., each position of the vector would store a byte, so, a datum that require four bytes to be stored would need four positions in the vector.

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

This ideas will be better understood by the use of a graphical example. Figure 5.1(a) shows the new structures that have to be implemented, and the initial values used in our example. In this way, suppose that a single thread is in execution (to avoid an unnecessarily complex situation) and it has managed two data, `c` and `b`, with a size of `8` and `4` respectively. So, taking into account that this library is implemented in `C` language (vectors begin at 0 position), actual offset of this thread will be `12`.

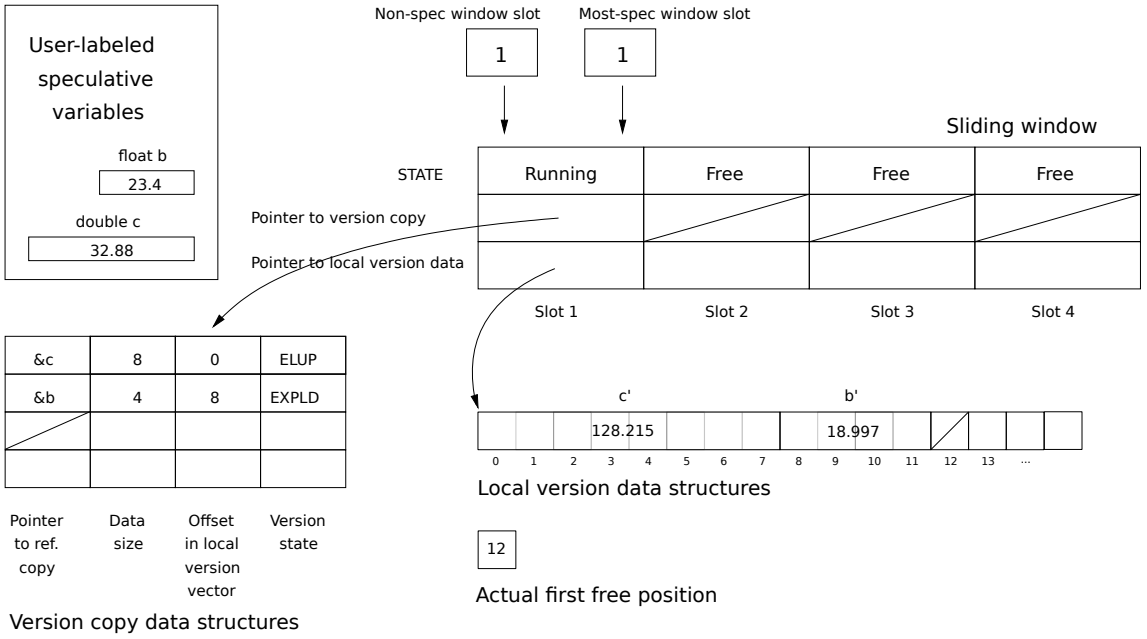
On the other hand, Figure 5.1(b), and Figures 5.2(c) and (d) show how a new data would be added to the version copy of a slot. At that moment, a new data is modified in the context of this thread: `a`, with a size of `1`. As it can be seen, this datum has not been used yet (see Figure 5.1(b)). Therefore should be added.

First of all, datum is stored in the new `Local Version Data` vector. After storing a copy of `a`'s value, thread working in slot 1 adds a new row to its version copy data structure, storing the address of `a`, its data size, the offset where is stored the version copy of `a` being managed by the thread, and the new state for this version copy, `MOD` (see Figure 5.2(c)).

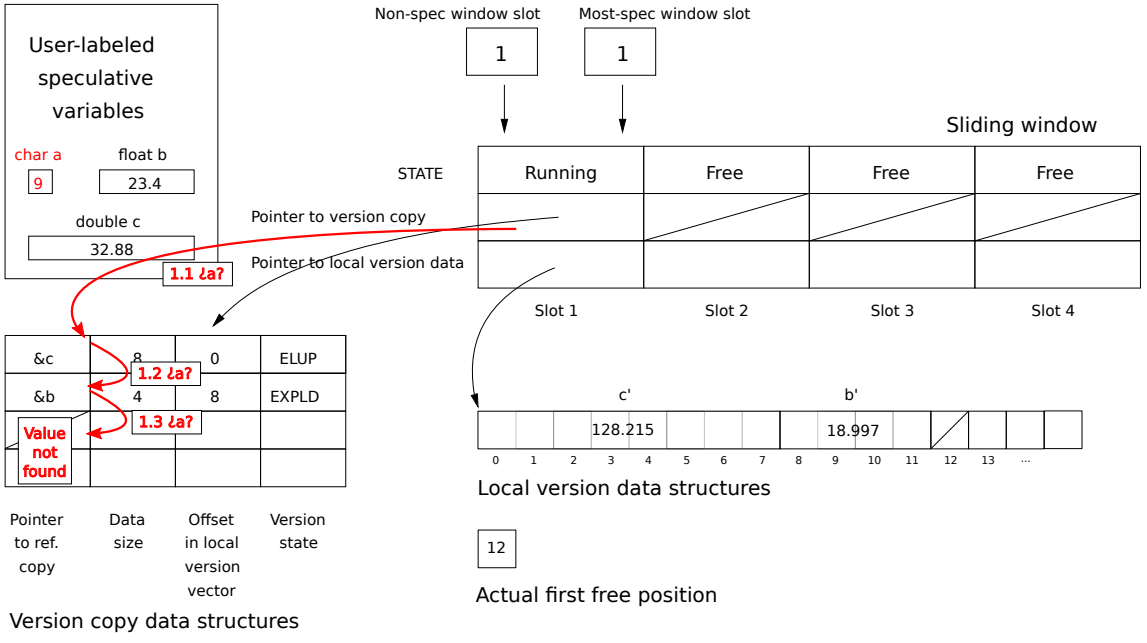
Finally, we should update the value of the first free position in the `Version Data` vector, in this case, we should augment this value by one: $12+1 = 13$ (see Figure 5.2(d)).

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

Therefore, this optimization avoids a sequential access to the elements of the version copy of the threads. Specifically, being $T$ the number of threads, and $N$ the number of data elements stored locally, this operation was initially in $T \times O(N) = O(TN)$. With the new scheme the free is done in $T \times O(1) = O(T)$, so, this optimization should theoretically improve the experimental results.

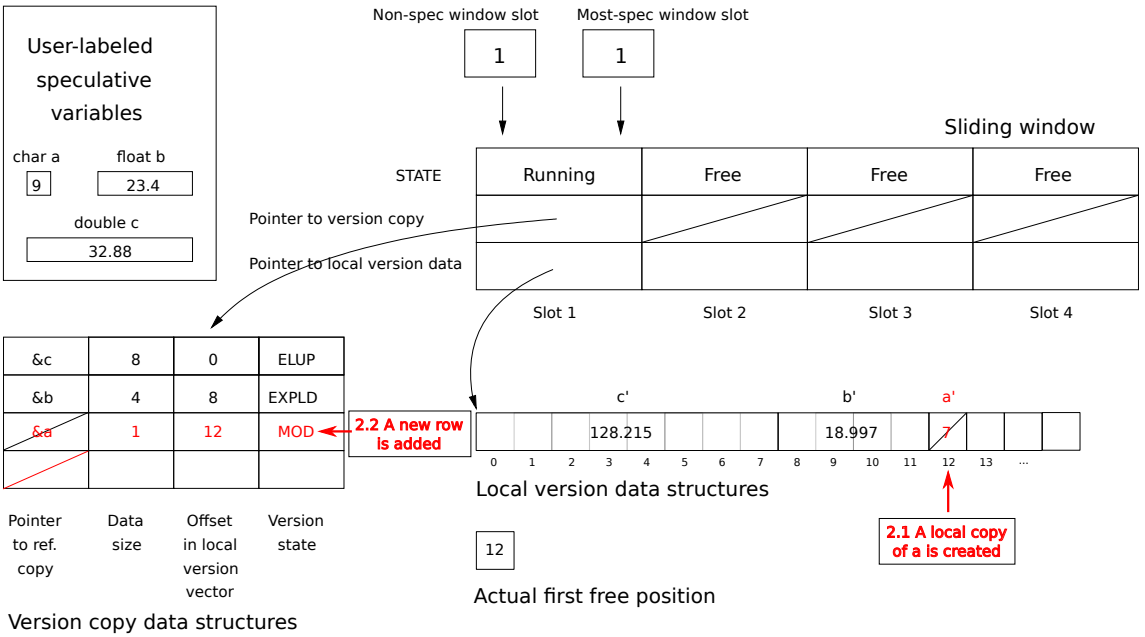



(a)

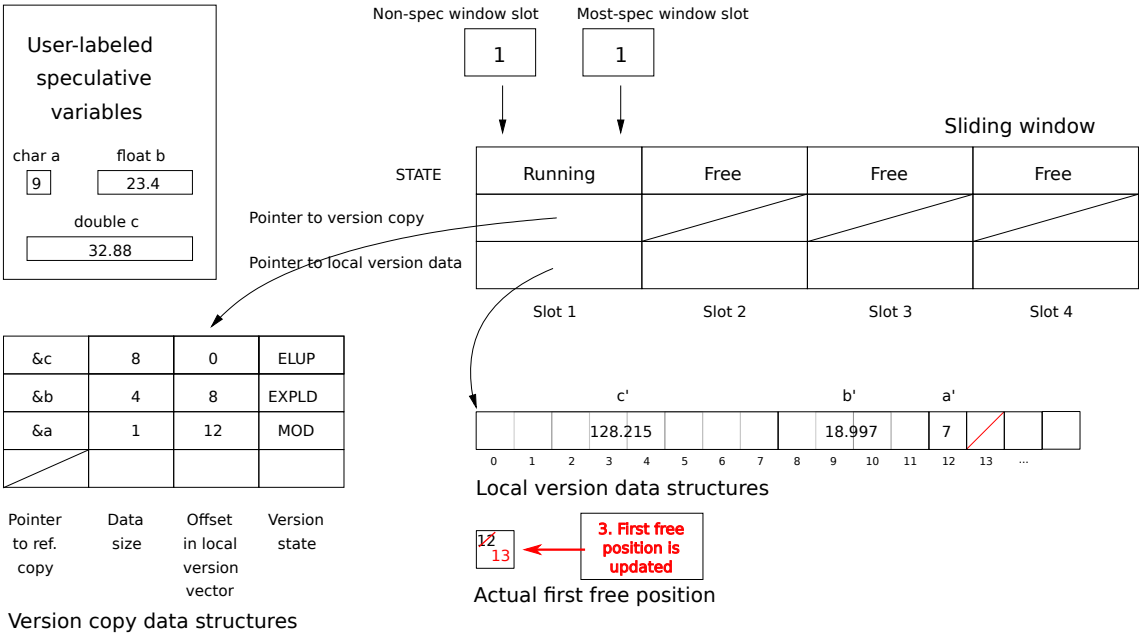



(b)

Figure 5.1: Optimization 1: Reducing operating system calls example (1/2). (a) Initial values of the example with the new data structures of this optimization of the speculative library. (b) Thread working in slot 1 scans its version copy to find the value.




(c)




(d)

Figure 5.2: Optimization 1: Reducing operating system calls example (2/2). (c) After creating a local copy of a in its vector of local copies, thread working in slot 1 adds a new row to its version copy data structure. (d) After storing a copy of a's value, thread working in slot 1 augment the indicator to the first position free in the vector of local copies.

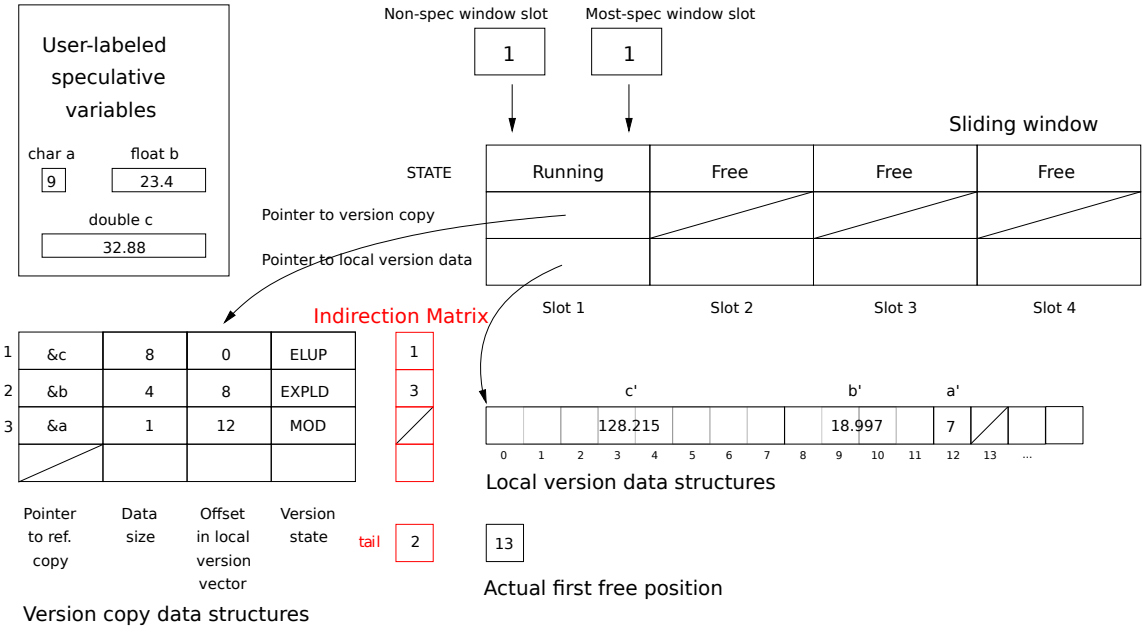



Figure 5.3: Optimization 2: Structures with the Indirection Matrix

## 5.3 Commit optimization

Commit operations in the original library required to check all the local elements accessed by the thread, both modified, and unmodified items.

So, in addition to the other modifications, we performed an optimization inspired in the work described in [10, 11], an ***Indirection Matrix***. This matrix aims to optimize commit operations because it will be used to store a list of updated elements, in order to only commit them.

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

Figure 5.3 shows this new structures in the general architecture, including the optimization of the vector with the local data. In the situation depicted in the example, imagine that thread finishes the execution of its chunk of iterations, as long as the thread is the non-spec one, it will begin to commit its elements. Therefore, it scans its `Indirection Matrix` to locate those variables in `ELUP` or `MOD` state. In our example, `c` has been loaded and updated, and `a` has been modified so it copies the content of `a'` into `a`, and the content of `c'` into `c`. In this way, the attempt of committing the content of `b` is avoided, unlike what happened in the original solution (see Figures 4.8, 4.9, 4.10, 4.11, 4.12 and 4.13).

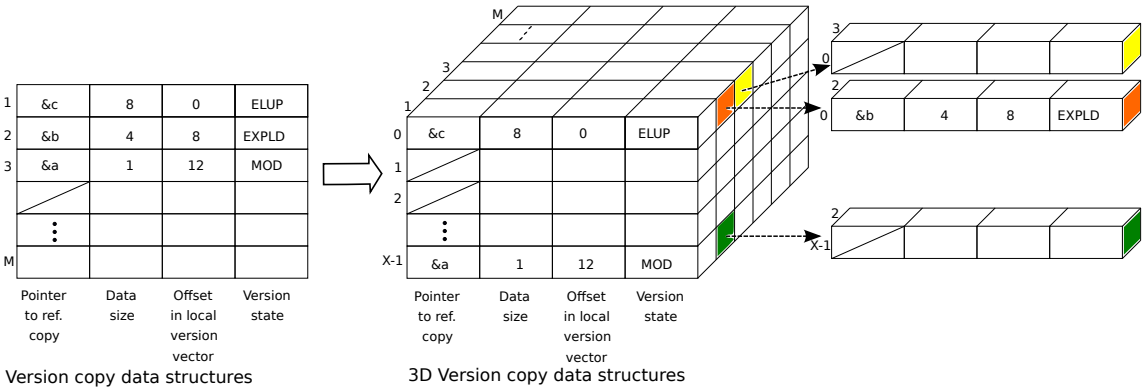



Figure 5.4: Optimization 3: Structures with three dimensions.

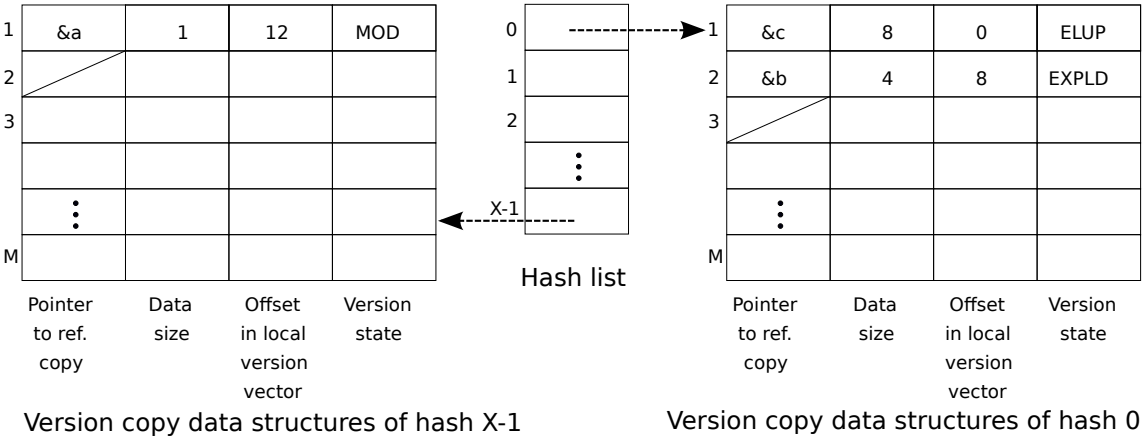



Figure 5.5: Optimization 3: Structures with three dimensions, another point of view.

## 5.4 Hash structure: Version Copy in Three Dimensions

We should not forget that the main bottleneck comes from the sequential checks performed during *specload()* and *specstore()* functions, where each element should be inspected to be compared with one of the arguments, to detect if it has been used before, or to get the most updated value of it.

One way to speed up these searches is to switch to a different data structure to hold local version copies of data. Instead of using a single table per thread as version copy data structure, we have developed an alternative structure with $X$ tables. Before accessing the data, 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 $H$th 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.

Figure 5.4 shows the new version copy structure in three dimensions with an example. Suppose that the previous version copy of a thread has the values depicted on the left of the figure. With the new structure in three dimensions, this version copy is transformed

into the structure shown on the right of the picture, supposing that the hash of `c` and `b` is `0`, and the hash of `a` is `X-1`. Also, the first value of each hash row that has not been previously used, is marked as void. The same occurs with the third position (depth) of the `0` hash position, and the second position (depth) of the `X-1` hash position.

In the practice, these ideas have been implemented with the mentioned third dimensional structure but, in order to have a better understanding of this optimization, another point of view could be used: Imagine the structure with three dimensions as if exists a vector with `H` positions where each one of them have `H` pointers to `H` version copy structures, i.e., instead of using a version copy for each slot, use `H` version copies for each slot. This idea, conceptually similar to the one explained in the previous paragraph and the same example that is shown in Figure 5.4, is depicted in Figure 5.5.

The concepts introduced by the first optimization continues being correct, so, a single `local version data` structure is used by each slot. In this way, variables used in Figures 5.4 and 5.5 preserve the same offset values in the version without the third optimization and in the new version.

On the other hand, the second optimization, that is, include an `Indirection Matrix` to the schema, can not be used without the application of some modifications to allow that the mentioned matrix has the same functionality than the previous one. The following subsection introduces the modifications performed.

### 5.4.1 Changes needed in the *Indirection Matrix*

As we stated in the previous paragraphs the new version of the speculative parallelization library uses an additional dimension to improve performance. Therefore, the *Indirection Matrix* also needs to add an additional dimension to its structure, because in the previous version of the scheme a single `version copy` was managed by each slot. Consequently, if no changes were performed in the mentioned matrix, each position of the *Indirection Matrix* will point to a single `version copy` instead of taking into account existing `H version copies`. On the other hand, perform changes in the basis of the *indirection matrix* by including the hash position `H` instead of the position of the datum in the version copy is not desirable because a hash position will point to several data, and updating one of them, its position would be added. By extension of this operation, all the row would be added, including those data from the row that have not been modified. So, in this way unmodified data would be committed.

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

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

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

## 5.5 Final structure of the speculative library

Figure 5.6 shows the scheme with all the modifications performed.

### 5.5.1 Implementation details

In order to take advantage of two and three-dimensional matrix solutions we decided to keep both types through the use of a compilation flag, namely `MOTOR_2D` or `MOTOR_3D`.

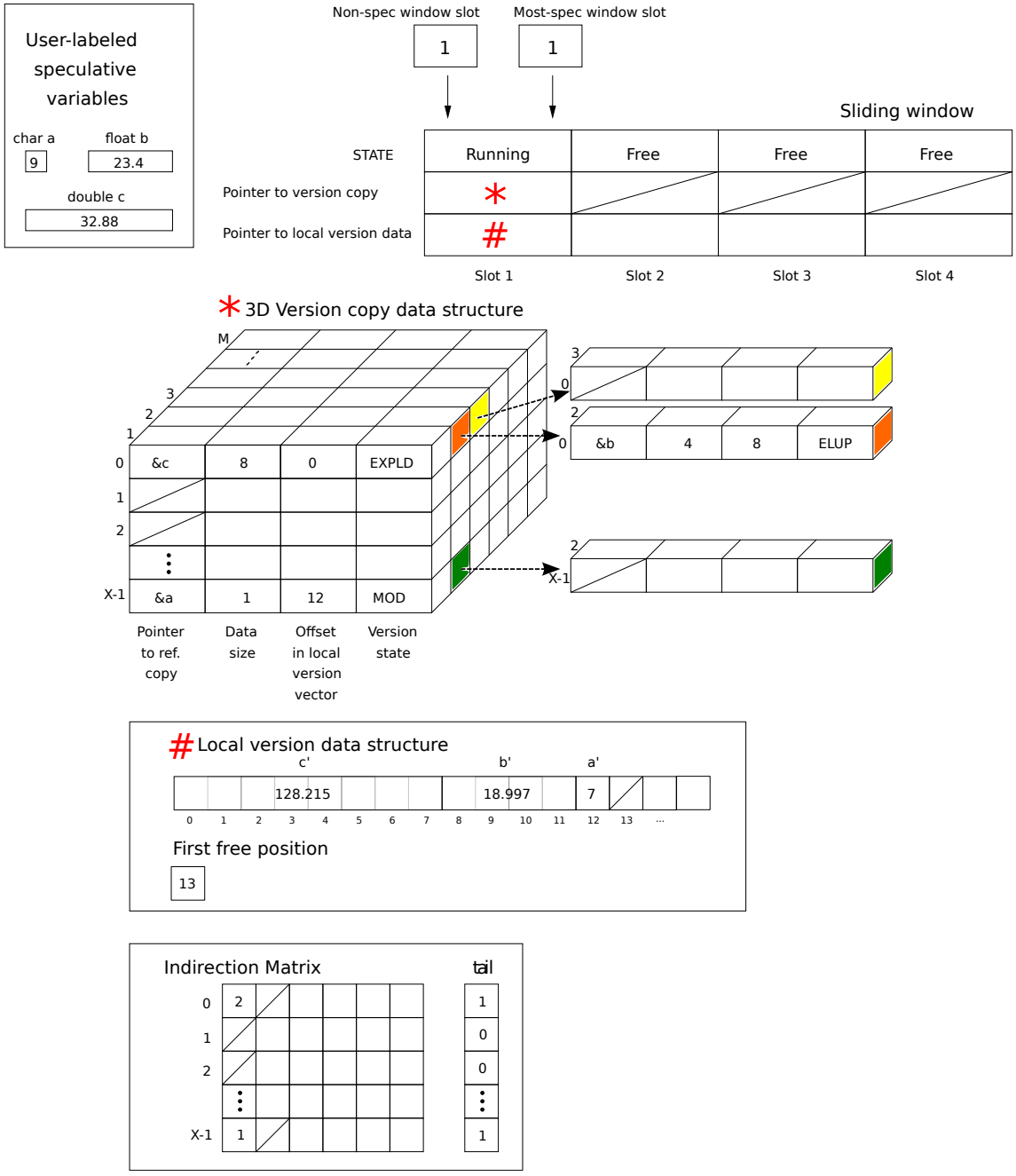



Figure 5.6: Structure of the speculative library after include all the optimizations.

# Chapter 6

# Experimental evaluation

*Para saber si las optimizaciones aplicadas a la librería de paralelización especulativa conducen a beneficios en el tiempo de ejecución se han desarrollado varios experimentos comparando los resultados obtenidos con la versión original disponible, respecto a las desarrolladas tras realizar las optimizaciones. Para ello se han utilizado dos tipos de benchmarks: aplicaciones utilizadas en el mundo real, y aplicaciones sintéticas para probar el motor.*

*La aplicaciones reales son un subconjunto de algoritmos incrementales aleatorios e incluyen el cierre convexo, el cálculo del menor círculo contenedor y la triangulación de Delaunay, para un conjunto de puntos bidimensional en los tres casos.*

*Por otro lado, hemos utilizado tres aplicaciones sintéticas: la primera de ellas, denominada 'complete', nos permite probar el uso de diferentes tipos de datos en la misma aplicación, así como a estructuras de datos complejas. La segunda de ellas, denominada 'tough', provoca un gran número de violaciones de dependencia. Por último, la aplicación 'fast' es un benchmark diseñado para generar ejecuciones mucho más rápidas que las versiones secuenciales.*

*Los resultados experimentales para el cálculo del menor círculo contenedor muestran que, en el servidor empleado, las ejecuciones con la versión del sistema que genera los mejores tiempos de ejecución, a partir de 12 procesadores generarán tiempos mejores que los de la versión secuencial, con un speedup de 1.14×. Además, en dicha versión, obtenemos resultados un 27.8% mejores que en la versión original.*

*Para el cálculo del cierre convexo, hemos utilizado dos conjuntos de datos diferentes. Así, se han obtenido diferentes resultados según el conjunto de datos empleado. Con el primero de ellos, un conjunto de puntos que distribuidos en forma de cuadrado, y en la versión que mejores resultados produce, hemos obtenido un speedup del 1.05× respecto a la versión secuencial. Sin embargo, la ganancia respecto a la versión original de la librería es mucho más elevada con una mejora del 421.4%. Con el segundo conjunto de datos estudiado, se obtienen speedups en la mejor de las versiones de hasta el 1.61× respecto a la versión secuencial, con una ganancia respecto a la*

*versión original del motor del 314.7%.*

*Para la triangulación de Delaunay hemos empleado dos conjuntos de datos, uno de 100 000 puntos, y otro de 1 000 000. Los resultados de la versión secuencial son siempre mejores que los de la paralela para ambos conjuntos de datos. La ganancia obtenida respecto a las versiones originales han sido del 23.7% para el conjunto de datos de 100 000 puntos, y del 42.6% para el otro. Además, esta aplicación nos permitió detectar errores en la librería, ya que algunas ejecuciones con un cierto número de procesadores, provocaban errores en la versión original. Sin embargo, estos errores han sido en su mayoría resueltos en la versión más reciente del motor especulativo, como se refleja en las tablas de resultados.*

*En cuanto a las aplicaciones sintéticas, cabe mencionar que lo que se busca con ellas no es tanto reflejar el tiempo de ejecución, como comprobar la solidez de nuestra solución. En este sentido, tanto las versiones 'complete', como 'tough' fueron exitosas ya que se ejecutaron correctamente. Por otro lado, la aplicación 'fast', además de no contener errores, genera ejecuciones más rápidas a partir de 2 procesadores en todas las versiones del motor, llegando a alcanzar, para la versión que mejores resultados genera, con 16 procesadores, una ganancia del 15.6× respecto a la versión secuencial. La ganancia de la versión actual respecto la original es mínima para esta aplicación, concretamente, un 1.2%.*

## 6.1 Benchmarks description

To test the speculative library, we have used both real-world and synthetic benchmarks. In order to obtain a better understanding of the experimental results, both of them are going to be explained.

### 6.1.1 Real-world benchmarks

First of all, we will briefly review a kind of algorithms called randomized incremental algorithms, because all the benchmarks tested in our experiments are included in this category.

Randomized incremental algorithms have been studied in depth in the context of Computational Geometry and Optimization. Their use have allowed the development of simple, easy-of-implement and efficient algorithms that solve several problems. For example, line segment intersection, Voronoi diagrams, triangulations of simple polygons, linear programming and many others.

In its most general formulation, the input set of a randomized incremental algorithm is a set of elements (can or can not be points) that are subjected to some operations to obtain a certain output. In these algorithms, generally a loop iterates over all the elements that are inserted following a random order. The execution of two iterations in two different processors simultaneously requires that no dependences exist between results calculated in the first iteration and values needed by the second iteration. These kind of algorithms present a common dependence pattern between the iterations of the loop (the following section describes in depth this question) independently of the problem to solve. Informally, it could be said that at the beginning of the execution, many of the inserted elements modify the solution that is being calculated iteration by iteration. However, as the execution progresses, less dependences appear, i.e., less elements modify the solution. Regarding complexity analysis, the expected complexity of a randomized incremental algorithms would normally be much lower than the complexity found in the worst case. Speculative parallelization is the most effective technique to execute in parallel this dependences distribution.

Problems addressed to obtain experimental results are:

- Welzl algorithm to calculate the “2-dimensional Minimum Enclosing Circle” problem (2D-Mec).
- Clarkson et al. algorithm to calculate the “2-dimensional Convex Hull” problem (2D-Hull).
- *Jump-and-Walk* strategy to calculate the “2-dimensional Delaunay Triangulation” problem (2D-DT).

#### Minimum enclosing circle

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 [79], that solves the problem in linear time. This algorithm starts with a circle of radius equal to zero

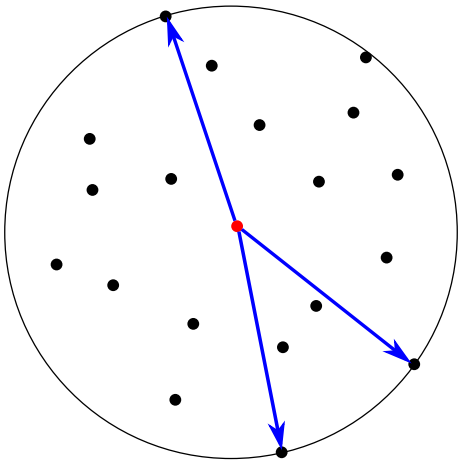


Figure 6.1: Minimum enclosing circle defined by three points.

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 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 can be 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.

Figure 6.1 shows an example of the minimum enclosing circle of a given input set.

### Convex hull

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.* [14] implementation [12, 36, 18]. The algorithm starts with a 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.

Figure 6.2 shows an example of the convex hull of a given input set.

### Delaunay triangulation

A triangulation is a subdivision of an area or plane polygon into a set of triangles, taking into account that each side of the triangle is shared by two adjacent triangles. Analogously a triangulation of a two-dimensional set of points is defined as a convex hull partition into triangles. The structure is a maximal family of disjoint interior triangles whose vertices are points of the set. Of course, there are not points located inside the triangles. Figure 6.3 shows that a single data set could generate different triangulations.

Delaunay triangulation [31] applied to a two-dimensional set of points affirms that a network of triangles is a Delaunay triangulation if all the circumcircles of all the triangles of the network are empty, i.e., the circumcircle of each triangle of the network contains

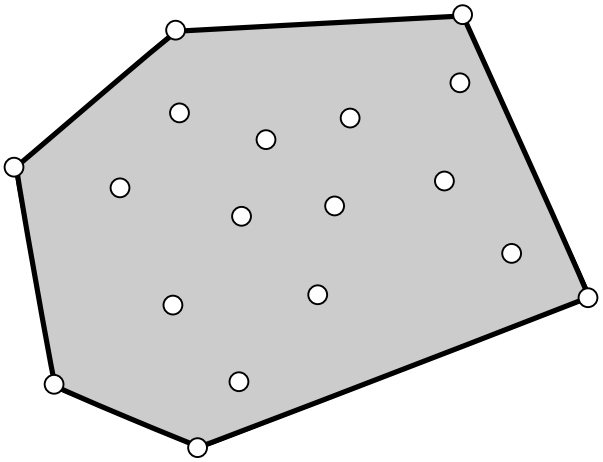


Figure 6.2: Convex hull of a set of points.

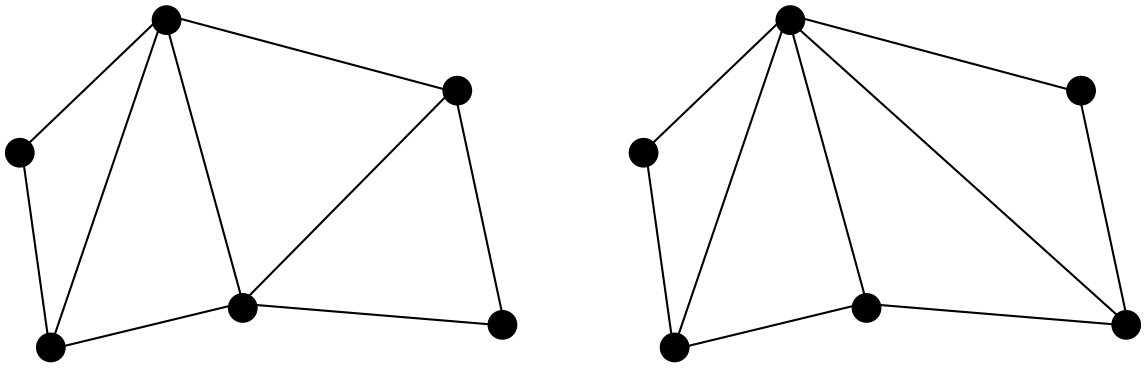


Figure 6.3: Two different triangulations with the same set of points.

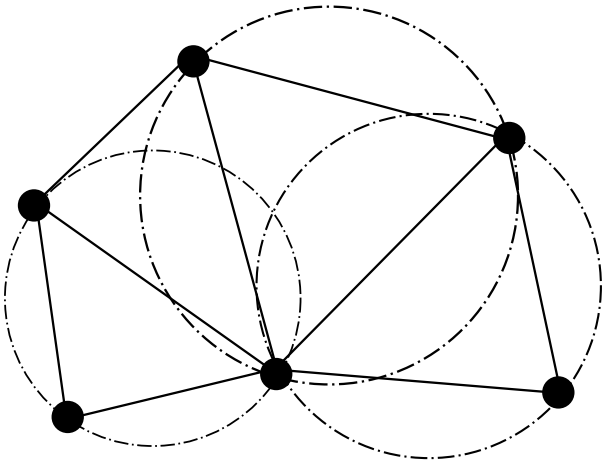


Figure 6.4: Delaunay triangulation of a set of points: Circumcircles of triangles shown do not contain any point inside them.

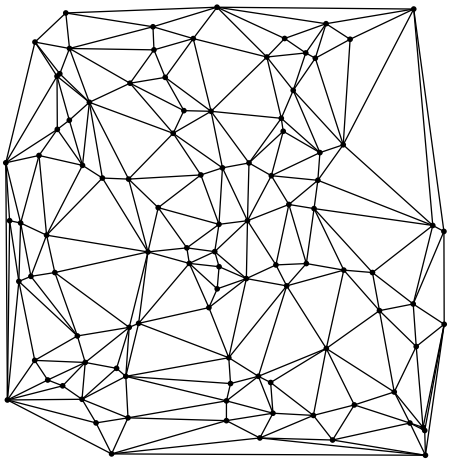


Figure 6.5: Delaunay triangulation of a set of 100 points.

no other vertices that those three that define the triangle. This condition ensures that the interior angles of the triangles are as large as possible and the length of the sides of the triangles is minimal. See Figure 6.4.

In this case, the randomized incremental construction of the Delaunay triangulation will be addressed by using *Jump-and-Walk* strategy, introduced by Mücke, Zhu *et al.* [59, 17]. This strategy uses 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 transverses 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 them updates the surrounding edges to keep the Delaunay properties.

Figure 6.5 shows an example of the Delaunay triangulation of a given input set that contains 100 points.

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

(a)

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

(b)

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

(c)

Figure 6.6: Synthetic benchmarks used: (a) Complete; (b) Tough; (c) Fast.

### 6.1.2 Synthetic benchmarks

Figure 6.6 shows the code of the three synthetic benchmarks used to perform the evaluation of our speculative library: Complete, Tough and Fast.

#### Complete

The Complete benchmark, shown in Figure 6.6(a), aims to concurrently test the most useful features of our solution, including (1) speculative access of data with different sizes, and (2) speculative access to data structures. While executing this loop in parallel, all the iterations lead to dependence violation.

#### Tough

The Tough benchmark, depicted in Figure 6.6(b), was designed to heavily test the robustness of our solution and of the underlying consistency protocol used. All of its iterations perform a load and a store on the same speculative data structure, with almost no computational load on private variables. This situation adversely affects performance, although the number of dependence violations during parallel execution is relatively small (4.46%).

#### Fast

The Fast benchmark, shown in Figure 6.6(c), has been designed to test the efficiency of the speculative scheduling mechanism. In this benchmark, only one of the 30 000 iterations (0.003%) lead to a dependence violation. Note that this single dependence is enough to prevent the compile-time parallelization of this loop.

## 6.2 Experimental results

### 6.2.1 Experimental environment

Experiments were carried out on an Intel S7000FC4URE server, equipped with four quad-core Intel Xeon MPE7310 processors at 1.6GHz and 32GB of RAM. The system runs Ubuntu Linux operating system. All threads had exclusive access to the processors during the execution of the experiments, and we used wall-clock times in our measurements. We have used `gcc 4.6.2` for all applications, with `-O4` optimization level. Times shown in the following sections represent the time spent in the execution of the parallelized loop for each application. The time needed to read the input set and the time needed to output the results have not been taken into account.

To achieve a better measure of runtime in all the applications we have used the average runtime of three executions all over experiments performed.

### 6.2.2 Engine version analyzed

Some different versions have been implemented in the process of development of the speculative library. The main versions developed are:

- **Original version (*v-40*):** This is considered the initial version of the library, that is, the version without any optimizations.
- **Indirection Matrix Version (*v-43*):** This version implements one of the described optimizations, the Indirection Matrix.
- **Version without system calls (*v-45a*):** This version incorporates the Indirection Matrix, and a second optimization, it removes all *malloc()* and *free()* functions related to memory allocation of local variables of the threads. During the development of this version we have encountered and fixed several issues that made versions *v-40* and *v-43* to crash when executing several benchmarks.
- **Three-dimensional Version (*v-45b*):** The last version used implements Indirection Matrices, removes all the functions mentioned in the previous point, and implements a three-dimensional structure in order to avoid the sequential access to all the local elements of the threads.

To obtain more information about the mentioned optimizations see Chapter 5.

### 6.2.3 Real-world benchmarks evaluation

#### Minimum enclosing circle

The set of points used to execute this application consists on 10 000 000 points. The block size used to perform the experiments with this application is 950 iterations per block.

With these experiments the two-dimensional structure produces the best results because this application does not use a big number of speculative variables, and then, it is not necessary to add a memory overload in the execution. So, speedups obtained with the three-dimensional version are too poor. Table 6.1 and Figure 6.7 show the results.

| Number of threads | Speed-up *v-40* | Speed-up *v-43* | Speed-up *v-45a* | Speed-up *v-45b* |
|---|---|---|---|---|
| 1 | 0.084 | 0.081 | 0.118 | 0.116 |
| 2 | 0.166 | 0.162 | 0.229 | 0.213 |
| 4 | 0.272 | 0.309 | 0.432 | 0.310 |
| 6 | 0.451 | 0.447 | 0.606 | 0.331 |
| 8 | 0.565 | 0.536 | 0.699 | 0.288 |
| 10 | 0.646 | 0.656 | 0.869 | 0.282 |
| 12 | 0.736 | 0.730 | **1.038** | 0.292 |
| 14 | 0.820 | 0.818 | **1.092** | 0.283 |
| 16 | 0.892 | 0.839 | **1.140** | 0.288 |

Table 6.1: Experimental results after the execution of the application that calculates the minimum enclosing circle of a set of points at Geopar server. The sequential time obtained was *0.633 seconds*.

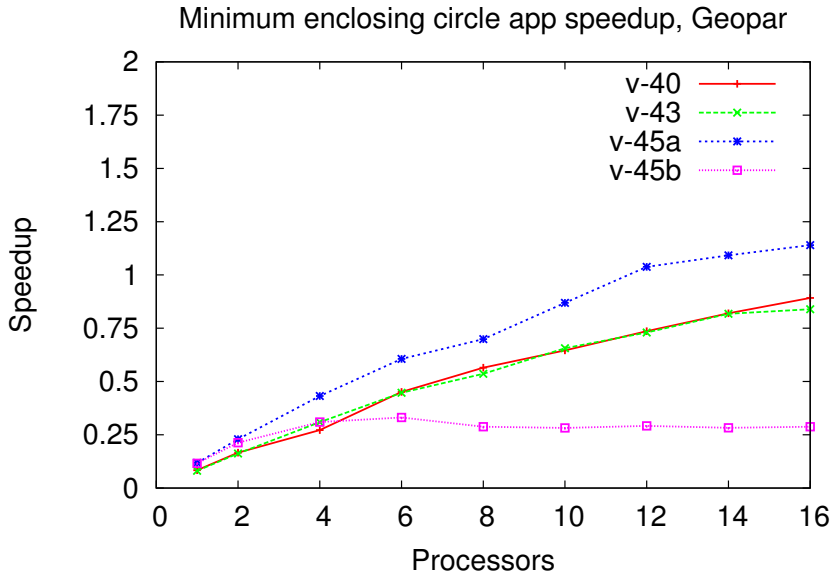



Figure 6.7: Speed-up obtained after executing 2D-MEC with the speculative library.

### Convex hull

In this case, two different set of points have been used, but all of them have 10 000 000 points, and then we obtain two results with this application. The block size used by this application has been different with each input set.

The three-dimensional structure produces the best results with both input sets. This application uses a big number of speculative variables, so sequential accesses to all of them was a big bottleneck. In this way, other versions have to perform accesses element by element, and their results are worse than the three-dimensional version (*v-45b*).

The two different input sets are described below, and the results obtained for each one are shown:

- **Square.in:** Contains a uniformly-distributed, square-shaped set of points. The block size used to perform the experiments with this input set has been of 4 000 iterations per block. Table 6.2 shows the experimental results of the application with this input set, and Figure 6.8 resumes the results obtained with this input set.

| Number of threads | Speed-up *v-40* | Speed-up *v-43* | Speed-up *v-45a* | Speed-up *v-45b* |
|---|---|---|---|---|
| 1 | 0.021 | 0.021 | 0.032 | 0.100 |
| 2 | 0.040 | 0.041 | 0.061 | 0.191 |
| 4 | 0.077 | 0.079 | 0.118 | 0.353 |
| 6 | 0.111 | 0.114 | 0.171 | 0.510 |
| 8 | 0.143 | 0.145 | 0.219 | 0.645 |
| 10 | 0.172 | 0.176 | 0.264 | 0.775 |
| 12 | 0.200 | 0.203 | 0.307 | 0.876 |
| 14 | 0.225 | 0.228 | 0.348 | 0.981 |
| 16 | 0.248 | 0.252 | 0.382 | **1.045** |

Table 6.2: Experimental results after execute the application that calculates the convex hull of a square-shaped input set at Geopar server. The sequential time obtained was *2.120 seconds*.

- **Kuzmin.in:** Contains a uniformly-distributed, disc-shaped set of points that follows a Kuzmin distribution. The block size used to perform the experiments with this input set has been of 11 000 iterations per block. Table 6.3 shows the experimental results of the application with this input set, and Figure 6.9 resumes the results obtained with this input set.

### Delaunay triangulation

With this application have been used two set of points with a different number of points. Specifically we have used 1 000 000 of points to test the speculative library, and then a subset of this that consists on 100 000 points. Block size used by this application is 1 iteration per block in both types of execution.

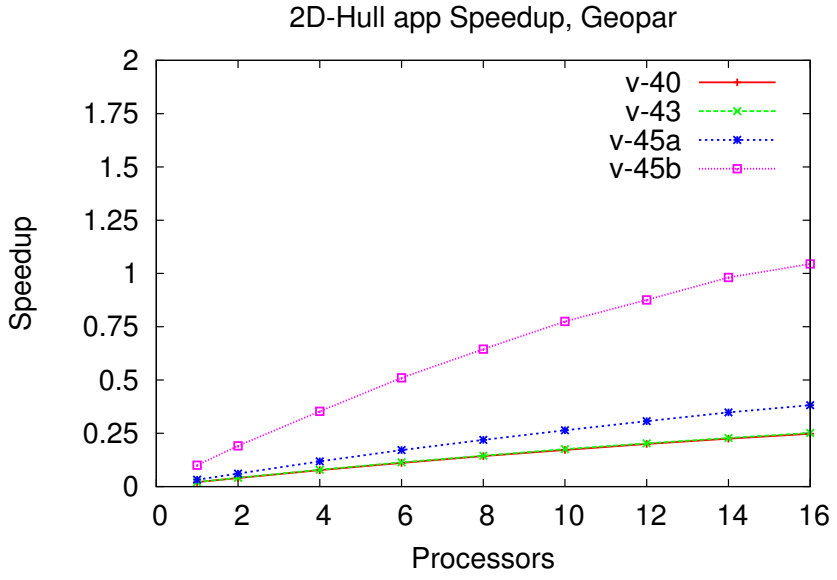



Figure 6.8: Speed-up obtained after executing 2D-Hull with the speculative library with the Square input set.

| Number of threads | Speed-up *v-40* | Speed-up *v-43* | Speed-up *v-45a* | Speed-up *v-45b* |
|---|---|---|---|---|
| 1 | 0.035 | 0.036 | 0.051 | 0.117 |
| 2 | 0.070 | 0.071 | 0.101 | 0.231 |
| 4 | 0.137 | 0.141 | 0.200 | 0.450 |
| 6 | 0.204 | 0.208 | 0.297 | 0.661 |
| 8 | 0.269 | 0.273 | 0.392 | 0.870 |
| 10 | 0.332 | 0.340 | 0.488 | **1.069** |
| 12 | 0.393 | 0.403 | 0.575 | **1.255** |
| 14 | 0.452 | 0.463 | 0.666 | **1.443** |
| 16 | 0.511 | 0.524 | 0.754 | **1.608** |

Table 6.3: Experimental results after execute the application that calculates the convex hull of a disc-shaped input set that follows a kuzmin distribution at Geopar server. The sequential time obtained was *1.652 seconds*.

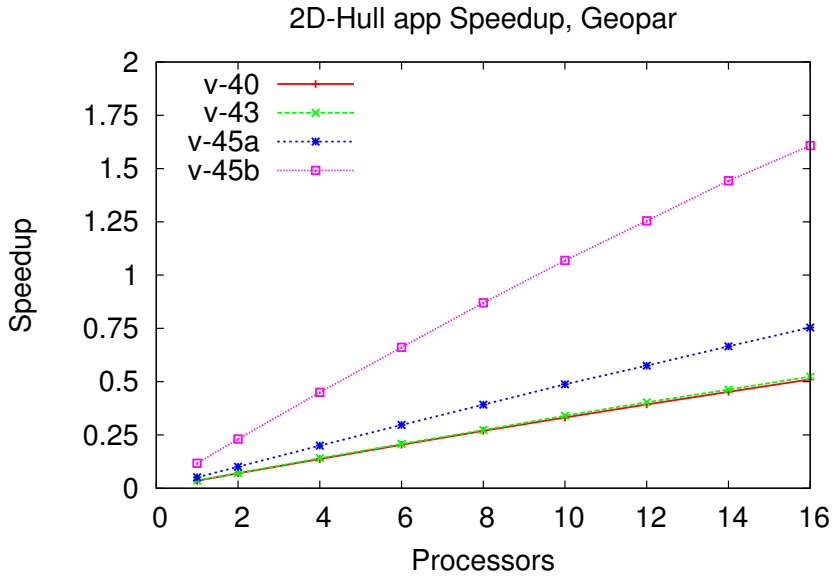



Figure 6.9: Speed-up obtained after executing 2D-Hull with the speculative library with the Kuzmin input set.

Here the two-dimensional structure produces the best results. This application uses a lot of speculative variables, however, its use is not synchronous, and then there are not many variables in use at the same time. However, there are not any version that leads to speedups higher than $1\times$.

Results obtained with the smallest input set used with this application are shown at Table 6.4. A resume of this results can be found at Figure 6.10.

Results obtained with the other input set used with this application are shown at Table 6.5. A resume of this results can be found at Figure 6.11.

### 6.2.4 Synthetic benchmarks evaluation

We have used the following synthetic examples in order to obtain some conclusions about the speculative library:

- *Complete* to use several different structures.
- *Tough* to induce many dependence violations, thus forcing the runtime library.
- *Fast* to obtain a high speed-up.

#### Complete

To execute this application we have use a set of points of 4 points and a block size of 500 iterations per block.

Table 6.6 shows the experimental results. In this table it can be seen that the speedup obtained is extremely poor. However, the main goal of the use of this kind of application

| Number of threads | Speed-up *v-40* | Speed-up *v-43* | Speed-up *v-45a* | Speed-up *v-45b* |
|---|---|---|---|---|
| 1 | 0.354 | 0.356 | 0.450 | 0.408 |
| 2 | 0.449 | 0.431 | 0.559 | 0.582 |
| 4 | 0.509 | 0.487 | 0.611 | 0.347 |
| 6 | 0.564 | 0.484 | 0.612 | 0.292 |
| 8 | 0.508 | 0.466 | 0.604 | 0.258 |
| 10 | 0.528 | 0.505 | 0.643 | 0.248 |
| 12 | 0.528 | 0.513 | 0.653 | 0.239 |
| 14 | 0.526 | 0.503 | 0.646 | 0.230 |
| 16 | – | – | 0.625 | 0.219 |

Table 6.4: Experimental results after execute the application that calculates Delaunay triangulation of an input set of 100 000 points at Geopar server. The sequential time obtained was *1.801 seconds*.

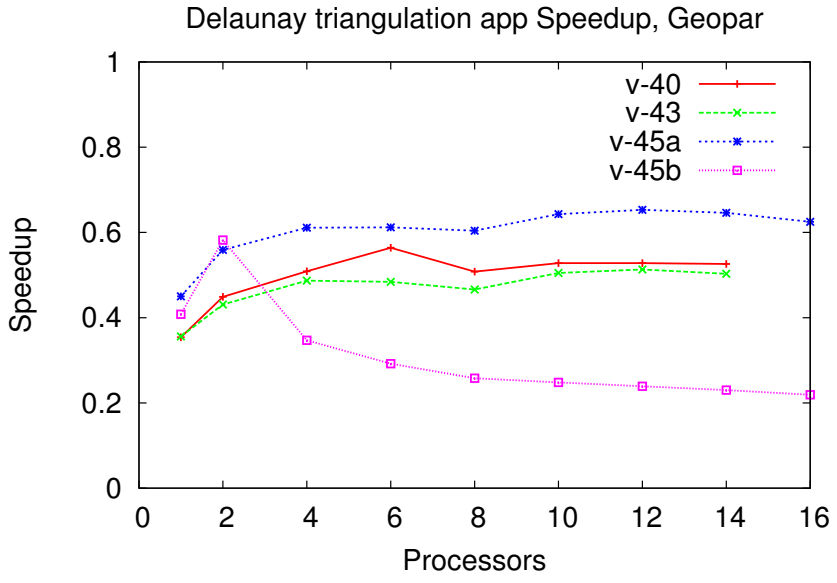



Figure 6.10: Speed-up obtained after executing Delaunay triangulation with the speculative library with the input set of 100 000 points.

| Number of threads | Speed-up *v-40* | Speed-up *v-43* | Speed-up *v-45a* | Speed-up *v-45b* |
|---|---|---|---|---|
| 1 | 0.292 | 0.287 | 0.362 | 0.399 |
| 2 | 0.341 | 0.337 | 0.493 | 0.470 |
| 4 | 0.383 | 0.375 | 0.480 | 0.423 |
| 6 | 0.394 | 0.349 | 0.510 | 0.331 |
| 8 | 0.378 | 0.361 | 0.517 | 0.291 |
| 10 | 0.397 | – | 0.540 | 0.278 |
| 12 | – | – | 0.552 | 0.270 |
| 14 | – | – | 0.566 | 0.257 |
| 16 | – | – | 0.557 | – |

Table 6.5: Experimental results after execute the application that calculates Delaunay triangulation of an input set of 1 000 000 points at Geopar server. The sequential time obtained was *21.946 seconds*.

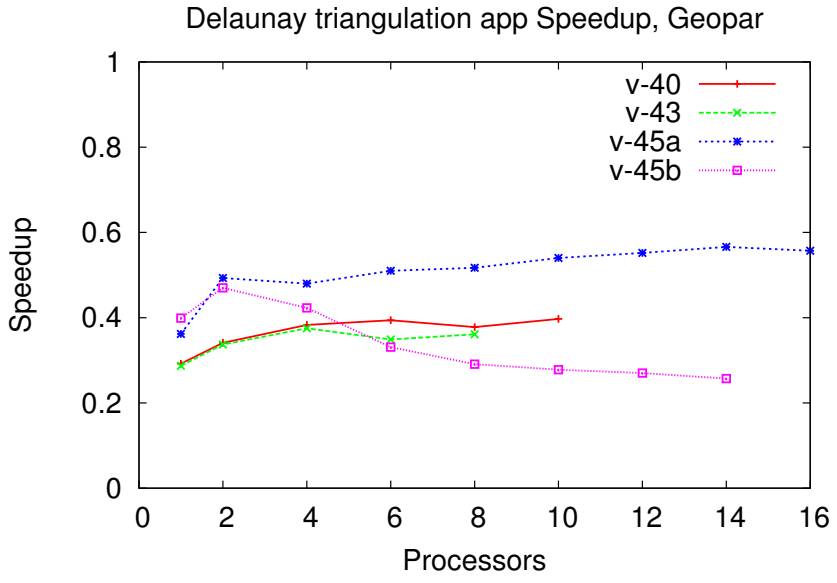



Figure 6.11: Speed-up obtained after executing Delaunay triangulation with the speculative library with the input set of 1 000 000 points.

is achieved: The parallel execution finishes successfully. Figure 6.12 depict the speed-up obtained with this application.

| Number of threads | Speed-up *v-40* | Speed-up *v-43* | Speed-up *v-45a* | Speed-up *v-45b* |
|---|---|---|---|---|
| 1 | 0.040 | 0.036 | 0.047 | 0.045 |
| 2 | 0.013 | 0.012 | 0.012 | 0.028 |
| 4 | 0.013 | 0.012 | 0.014 | 0.017 |
| 6 | 0.013 | 0.013 | 0.015 | 0.015 |
| 8 | 0.013 | 0.013 | 0.014 | 0.013 |
| 10 | 0.013 | 0.012 | 0.013 | 0.014 |
| 12 | 0.014 | 0.011 | 0.014 | 0.014 |
| 14 | 0.012 | 0.012 | 0.013 | 0.014 |
| 16 | 0.013 | 0.011 | 0.013 | 0.013 |

Table 6.6: Experimental results after execute the complete synthetic application at Geopar server. The sequential time obtained was *0.605 seconds*.

### Tough

To execute this application we have use a set of points of 100 points. Also, a block size of 100 iterations per block has been used.

Table 6.7 shows the experimental results. This table shows that the speedup obtained continues being very low. However, the main goal of the use of this kind of application is achieved: The parallel execution finishes successfully.

| Number of threads | Speed-up *v-40* | Speed-up *v-43* | Speed-up *v-45a* | Speed-up *v-45b* |
|---|---|---|---|---|
| 1 | 0.074 | 0.071 | 0.108 | 0.129 |
| 2 | 0.040 | 0.037 | 0.050 | 0.095 |
| 4 | 0.020 | 0.019 | 0.025 | 0.022 |
| 6 | 0.016 | 0.015 | 0.021 | 0.019 |
| 8 | 0.015 | 0.013 | 0.019 | 0.016 |
| 10 | 0.013 | 0.012 | 0.017 | 0.013 |
| 12 | 0.011 | 0.010 | 0.015 | 0.013 |
| 14 | 0.010 | 0.009 | 0.014 | 0.011 |
| 16 | 0.009 | 0.008 | 0.012 | 0.010 |

Table 6.7: Experimental results after execute the tough synthetic application at Geopar server. The sequential time obtained was *0.031 seconds*.

Figure 6.13 depicts the speed-up obtained with this application.

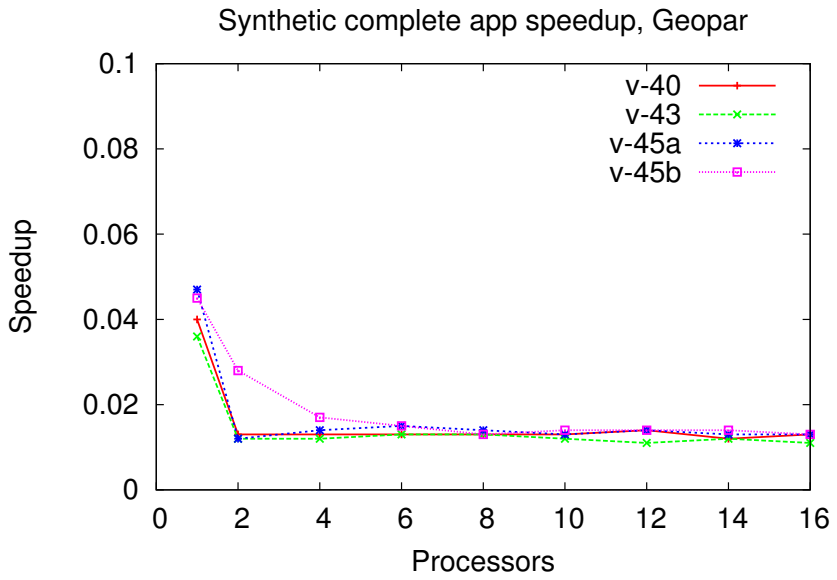



Figure 6.12: Speed-up obtained after executing complete synthetic example with the speculative library.

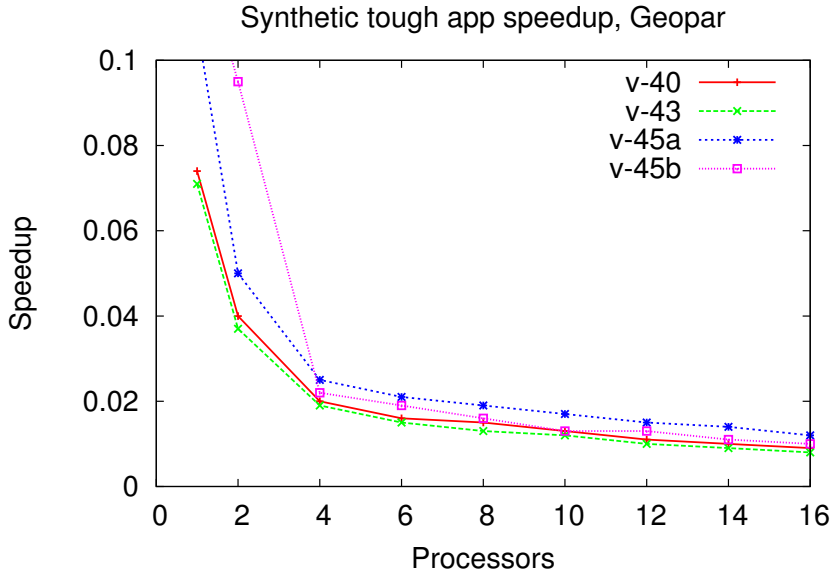



Figure 6.13: Speed-up obtained after executing tough synthetic example with the speculative library.

**Fast**

To execute this application we have use a set of points of 4 points. Furthermore, a block size of 500 iterations per block has been used.

This application produces the best results with the two-dimensional structure of the library, that is, the version *v-45a*.

In Table 6.8 can be found the experimental results obtained. Results of all versions show that this benchmark produces speedups higher than $1\times$ with all the executions that use more than one processor. The parallel execution of this benchmark with 16 processors leads to a $15.16\times$ speedup with the version *v-45a*. The obtained efficiency, $94.75\%$ for 16 threads, indicates that the overhead due to the speculative scheduling mechanism itself is negligible.

| Number of threads | Speed-up *v-40* | Speed-up *v-43* | Speed-up *v-45a* | Speed-up *v-45b* |
|---|---|---|---|---|
| 1 | 1.000 | 0.992 | **1.005** | **1.009** |
| 2 | **2.987** | **2.704** | **3.016** | **3.138** |
| 4 | **5.488** | **5.035** | **5.800** | **5.803** |
| 6 | **7.763** | **8.037** | **9.126** | **9.083** |
| 8 | **9.195** | **8.555** | **10.007** | **9.998** |
| 10 | **10.553** | **10.354** | **12.883** | **12.543** |
| 12 | **12.525** | **11.402** | **14.574** | **14.627** |
| 14 | **13.902** | **12.513** | **15.128** | **15.126** |
| 16 | **14.981** | **13.587** | **15.161** | **14.833** |

Table 6.8: Experimental results after execute the fast synthetic application at Geopar server. The sequential time obtained was *4.467 seconds*.

Figure 6.14 depicts the speed-up obtained with this application.

## 6.3 General evaluation of the results

Experimental results in terms of execution time clearly show that the improvements applied to the library have a direct repercussion on performance. All applications have better execution times than those obtained in the previous version [23]. In this way,

However, results continue being worse than those achieved by Cintra and Llanos in its purpose [10]. We could highlight the following reasons of this:

- One of the main problems is the inherent complexity that entails this development: New library works correctly, however, we are comparing the performance of a relatively new software, with another that have been developed, maintained and optimized during more than ten years. So, it is easy to think that this new speculative library could be substantially improved.

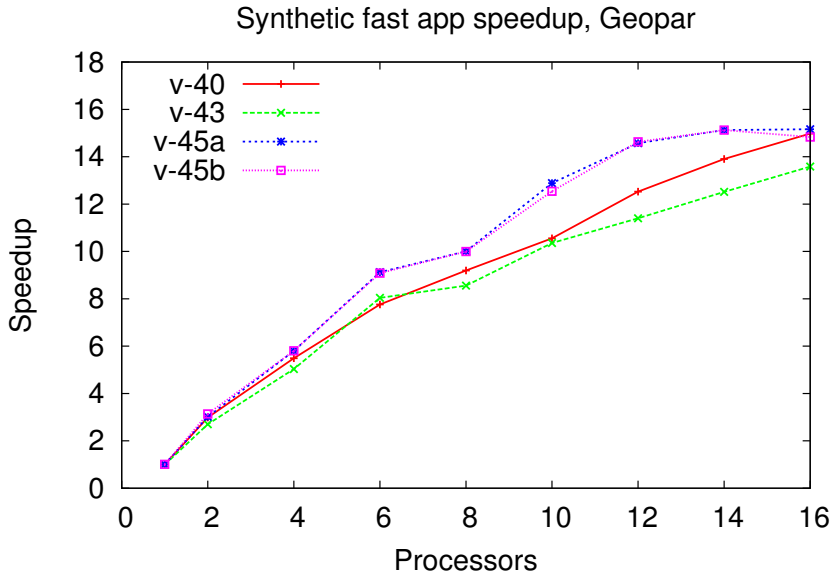



Figure 6.14: Speed-up obtained after executing fast synthetic example with the speculative library.

- Second, unlike Cintra and Llanos solution, new speculative engine is designed to execute speculatively the general case over random structures, not over a particular case. Consequently, we have lost performance over specific cases.

# Chapter 7

# Conclusions and future work

*El trabajo presentado ha consistido en las siguientes fases:*

1. *Se ha introducido la paralelización especulativa, describiendo el Estado del Arte, tanto en lo que respecta al trabajo realizado por grupos de todo el mundo como a las contribuciones del grupo Trasgo.*
2. *Para poner en perspectiva el trabajo realizado, se ha detallado:*
   (a) *El funcionamiento del motor base de ejecución especulativa desarrollado por Cintra y Llanos.*
   (b) *El funcionamiento de un nuevo motor que soluciona las principales limitaciones del anterior, y que fue desarrollado por el autor de este Trabajo de Fin de Máster en su anterior Trabajo de Fin de Grado.*
3. *Gracias en parte al estudio del Estado del Arte, se han localizado los principales cuellos de botella en el motor mencionado anteriormente, y se han enumerado soluciones que podrían mejorar su rendimiento.*
4. *Se han implementado dichas mejoras en el motor.*
5. *Se han obtenido resultados experimentales que constatan las mejoras producidas por las optimizaciones descritas.*

*Tras la finalización de este trabajo podemos afirmar que:*

- *El análisis del Estado del Arte nos permite concluir que el uso de técnicas de paralelización especulativa, aún teniendo en cuenta las mejoras introducidas, no produce mejoras en el rendimiento de las aplicaciones secuenciales si el número de violaciones de dependencias es demasiado alto. Por el contrario, si hay pocas dependencias, la paralelización especulativa generalmente producirá buenos resultados.*
- *Se ha confirmado que los principales cuellos de botella se producían debido a los accesos secuenciales a las matrices de datos locales producidos por cada llamada a las operaciones de lectura y escritura especulativas.*
- *Optimizaciones como la reducción de llamadas al sistema para reservar y liberar memoria, o la sustitución de estructuras de datos para evitar*

*el recorrido secuencial de elementos, conllevan mejoras reales en los tiempos de ejecución.*

- *El trabajo realizado ha dado lugar a que la nueva versión del motor especulativo permita la ejecución de aplicaciones en paralelo más rápidamente que sus correspondientes ejecuciones secuenciales.*
- *Pese a las mejoras introducidas, el motor original de Cintra y Llanos, aunque no dispone de las funcionalidades requeridas para ser puesto en producción, sigue siendo más rápido que el nuestro, aunque confiamos en alcanzar su rendimiento en un futuro próximo.*

*Como principales vías para la continuación de este TFM podríamos destacar: aplicar técnicas de squash exclusivo, mejorar aún más las estructuras de datos utilizadas para optimizar los recorridos, implementar operaciones de reducción, optimizar la memoria empleada por el motor, exportar la librería a un entorno de GPUs o aplicar técnicas de predicción de valores. Todo esto tiene como objetivo explotar al máximo las posibilidades de paralelizar aplicaciones que a priori no podrían paralelizarse, y obtener tiempos de ejecución mucho menores.*

*Finalmente, cabe decir que este trabajo forma parte de un artículo científico denominado "OpenMP meets TLS", enviado el día 17 de julio de 2013 al congreso ASPLOS (Architectural Support for Programming Languages and Operating Systems), considerado el congreso más importante del mundo en este ámbito, y calificado según la escala CORE 2008 como A*.*

## 7.1 Summary

The work carried out in this Master Thesis consists in the following topics:

1. We have presented the field of speculative parallelization of applications, showing the State of the Art, not only with respect to the work of the main research groups, but also showing the contributions of the Trasgo research group. This allowed us to put our own work in the field in a wider perspective.

2. In order to gain perspective of the work carried out in this Master Thesis:

   (a) We have described the behavior of the original, speculative parallelization engine developed by Cintra and Llanos in 2003.

   (b) We have shown the behavior of the new speculative engine that overcomes the limitations of the former one, developed by the author of this Master Thesis as part of his B.Sc. Thesis in 2012.

3. The main disadvantage of this speculative library was its poor performance. Thanks in part to the related work, we have isolated the main bottlenecks of our system, and we found solutions that we expected to improve the library performance.

4. We have implemented these solutions in the library.

5. The experimental results obtained, both with real-world and synthetic applications, show the correctness of the solutions proposed.

## 7.2 Conclusions

As the main conclusions of this Master Thesis we can say that:

- The study of the State of the Art led us to the conclusion that the use of speculative parallelization techniques, even taking into account the improvements introduced in this work, does not always lead to a better performance, in particular when the number of dependence violations issued at runtime is high.

- We have correctly predicted that the main bottlenecks of the system was due to the sequential accesses to local data structures, provoked by calls to the speculative load and store operations.

- Optimizations such as the reduction in the number of memory management system calls, and the replacement of data structures to avoid their sequential traversing, lead to real improvements in the execution time.

- Finally, the work carried out during this Master Thesis leads to a new version of the speculative parallelization library that effectively speeds up the running time of sequential applications.

- Despite of the work carried out so far, the original Cintra and Llanos' library is still faster, although we are confident in being able to reach that figures in the near future.

## 7.3 Future work

The future work is related to add to our system some of the techniques described in the State of the Art, and create new ones. In particular:

- One of the best optimizations applied to the original speculative library was the incorporation of *exclusive squashing* techniques. The original squash implementation, called *inclusive squashing*, consists on discarding the offending thread and *all* its successors. Exclusive squashing, on the other hand, only discards threads that have consumed "polluted" data [28].
- To optimize the store of the pointer addresses. Right now, addresses are stored with the use of integers with a size of 4 or 8 bytes. With this new optimization, if an address value is higher than pointer size, it would be stored. On the other hand, if the size do not achieve the pointer size, it would directly save the datum.
- Original architecture supports sum and calculation of the maximum operations, however these improvements are not applied to the new engine.
- Apply some of the ideas located in [85] to achieve an architecture that supports GPU-TLS and pointer-based applications.
- Another possible way to improve the actual library will be the implement a "value prediction" in the same way that Raman et al. did in [67].
- Use the structure described in [83] to decrease memory overheads of the current library.

## 7.4 Publications

- Álvaro Estébanez, Diego R. Llanos, Arturo González-Escribano. Desarrollo de un motor de paralelización especulativa con soporte para aritmética de punteros. In *Proceedings of the XXIII Jornadas de Paralelismo*, Elche, September 2012. **Published**.
- Sergio Aldea, Álvaro Estébanez, Diego R. Llanos, Arturo González-Escribano. OpenMP meets TLS. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS 19, Salt Lake City, Utah, March 2014. **Submitted**

# Appendix A

# CD-Rom contents

*En este apéndice se recoge el contenido del CD adjunto a la documentación.*

This appendix serves to describes contents of the digital support attached to this documentation. The CD-ROM are structure in several directories:

- **Memoria**: It contains a PDF copy of this documentation.

# Bibliography


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

[2] Intel architecture set extensions programming reference. http://software.intel.com/file/36945, 2013.

[3] Yehuda Afek, Amir Levy, and Adam Morrison. Programming with hardware lock elision. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, pages 295–296, New York, NY, USA, 2013. ACM.

[4] Sergio Aldea, Diego R. Llanos Ferraris, and Arturo González-Escribano. Extending a source-to-source compiler with xml capabilities. In *Proceedings of the XXI Jornadas de Paralelismo*, Valencia, Spain, September 2010.

[5] Sergio Aldea, Diego R. Llanos Ferraris, and Arturo González-Escribano. Towards a compiler framework for thread-level speculation. In *PDP*, pages 267–271, 2011.

[6] Luiz André Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 282–293, New York, NY, USA, 2000. ACM.

[7] Mikael Berndtsson, Jörgen Hansson, Björn Olsson, and Björn Lundell. *Thesis Projects, A Guide for Students in Computer Science and Information Systems*. Springer, 2nd edition, October 2007. ISBN 978-1848000087.

[8] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, 1 edition, October 2000.

[9] Tong Chen, Jin Lin, Xiaoru Dai, Wei-Chung Hsu, and Pen-Chung Yew. Data dependence profiling for speculative optimizations. In Evelyn Duesterwald, editor, *Compiler Construction*, volume 2985 of *Lecture Notes in Computer Science*, pages 57–72. Springer Berlin Heidelberg, 2004.

[10] Marcelo Cintra and Diego R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *Proceedings of the SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, June 2003.

[11] Marcelo Cintra and Diego R. Llanos. Design space exploration of a software speculative parallelization scheme. *IEEE Trans. on Paral. and Distr. Systems*, 16(6):562–576, June 2005.

[12] Marcelo Cintra, Diego R. Llanos, and Belén Palop. Speculative parallelization of a randomized incremental Convex Hull algorithm. In *ICCSA 2004: Proc. Intl. Conf. on Computer Science and its Applications*, pages 188–197, Perugia, Italy, May 2004. Springer-Verlag LNCS 3045, ISSN 0302-9743.

[13] Marcelo Cintra, José F. Martínez, and Josep Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proc. of the 27th intl. symp. on Computer architecture (ISCA)*, pages 256–264, June 2000.

[14] K. L. Clarkson, K. Mehlhorn, and R. Seidel. Four results on randomized incremental constructions. *Comput. Geom. Theory Appl.*, 3(4):185–212, 1993.

[15] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, March 1998.

[16] Francis Dang, Hoo Yu, and Lawrence Rauchwerger. The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In *Proc. of the 16th International Parallel and Distributed Processing Symposium (IPDPS '02)*, April 2002.

[17] L. Devroye, E. P. Mücke, and Binhai Zhu. A note on point location in delaunay triangulations of random points. *Algorithmica*, 22(4):477–482, 1998.

[18] Pedro Díaz, Diego R. Llanos, and Belén Palop. Parallelizing 2D-convex hulls on clusters: Sorting matters. In *Proc. XV Jornadas de Paralelismo*, pages 247–252, Almería, Spain, September 2004. ISBN 84-8240-714-7.

[19] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. Software behavior oriented parallelization. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 223–234, New York, NY, USA, 2007. ACM.

[20] Jialin Dou and Marcelo Cintra. Compiler estimation of load imbalance overhead in speculative parallelization. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 203–214, Washington, DC, USA, 2004. IEEE Computer Society.

[21] Jialin Dou and Marcelo Cintra. A compiler cost model for speculative parallelization. *ACM Trans. Archit. Code Optim.*, 4(2), June 2007.

[22] Zhao-Hui Du, Chu-Cheow Lim, Xiao-Feng Li, Chen Yang, Qingyu Zhao, and Tin-Fook Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 71–81, New York, NY, USA, 2004. ACM.

[23] Álvaro Estébanez López, Diego R. Llanos, and Arturo González-Escribano. Desarrollo de un motor de paralelización especulativa con soporte para aritmética de punteros. Trabajo de fin de grado, Universidad de Valladolid, July 2012.

[24] Álvaro Estébanez López, Diego R. Llanos, and Arturo González-Escribano. Desarrollo de un motor de paralelización especulativa con soporte para aritmética de punteros. In *Proceedings of the XXIII Jornadas de Paralelismo*, Elche, Alicante, Spain, September 2012.

[25] Álvaro Estébanez López, Diego R. Llanos, and Arturo González-Escribano. Paralelización especulativa de un algoritmo para el menor círculo contenedor. Proyecto de fin de carrera, Universidad de Valladolid, September 2011.

[26] Min Feng, Rajiv Gupta, and Iulian Neamtiu. Effective parallelization of loops in the presence of i/o operations. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 487–498, New York, NY, USA, 2012. ACM.

[27] Lin Gao, Lian Li, Jingling Xue, and Pen-Chung Yew. Seed: A statically-greedy and dynamically-adaptive approach for speculative loop execution. *IEEE Transactions on Computers*, 62(5):1004–1016, 2013.

[28] Álvaro García-Yágüez, Diego R. Llanos, and Arturo González-Escribano. Exclusive squashing for thread-level speculation. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 275–276, New York, NY, USA, 2011. ACM.

[29] Alvaro Garcia-Yaguez, Diego R. Llanos, and Arturo Gonzalez-Escribano. Robust thread-level speculation. In *Proceedings of the 2011 18th International Conference on High Performance Computing*, HIPC '11, pages 1–11, Washington, DC, USA, 2011. IEEE Computer Society.

[30] Alvaro Garcia-Yaguez, Diego R. Llanos, and Arturo Gonzalez-Escribano. Squashing alternatives for software-based speculative parallelization. *IEEE Transactions on Computers*, 99(PrePrints):1, 2013.

[31] Alvaro Garcia-Yaguez, Diego R. Llanos, David Orden, and Belen Palop. Paralelización especulativa de la triangulación de delaunay. In *Proc. XX Jornadas de Paralelismo*, A Coruña, Spain, September 2009.

[32] Alok Garg, Raj Parihar, and Michael C. Huang. Speculative parallelization in decoupled look-ahead. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 413–423, Washington, DC, USA, 2011. IEEE Computer Society.

[33] María Jesús Garzarán, Milos Prvulovic, José María Llabería, Víctor Viñals, Lawrence Rauchwerger, and Josep Torrellas. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. *ACM Trans. Archit. Code Optim.*, 2(3):247–279, September 2005.

[34] Arturo González-Escribano and Diego R. Llanos. Paralelización especulativa y sus alternativas. *Actas XVIII Jornadas de Paralelismo*, 2007.

[35] Arturo González-Escribano, Diego R. Llanos, David Orden, and Belén Palop. Ejecución paralela de algoritmos incrementales aleatorizados. In Francisco Santos and David Orden, editors, *Proc. XI Encuentros de Geometría Computacional*, pages 79–86, Santander, Spain, June 2005. ISBN 84-8102-963-7.

[36] Arturo González-Escribano, Diego R. Llanos, David Orden, and Belén Palop. Parallelization alternatives and their performance for the convex hull problem. *Applied Mathematical Modelling, special issue on Parallel and Vector Processing in Science and Engineering*, 30(7):563–577, July 2006. ISSN 0307-904X.

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

[38] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. The stanford hydra cmp. *IEEE Micro*, 20(2):71–84, March 2000.

[39] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Proc. of the 8th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 58–69, 1998.

[40] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 102–, Washington, DC, USA, 2004. IEEE Computer Society.

[41] Ben Hertzberg and Kunle Olukotun. Runtime automatic speculative parallelization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 64–73, Washington, DC, USA, 2011. IEEE Computer Society.

[42] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 59–70, New York, NY, USA, 2004. ACM.

[43] Chuanle Ke, Lei Liu, Chao Zhang, Tongxin Bai, Brian Jacobs, and Chen Ding. Safe parallel programming using dynamic dependence hints. In *OOPSLA'11 Proceedings*, pages 243–258. ACM, 2011.

[44] Arun Kejariwal, Xinmin Tian, Milind Girkar, Wei Li, Sergey Kozhukhov, Utpal Banerjee, Alexander Nicolau, Alexander V. Veidenbaum, and Constantine D. Polychronopoulos. Tight analysis of the performance potential of thread speculation using spec cpu 2006. In *Proceedings of the 12th ACM SIGPLAN symposium on*

*Principles and practice of parallel programming*, PPoPP '07, pages 215–225, New York, NY, USA, 2007. ACM.

[45] Arun Kejariwal, Xinmin Tian, Wei Li, Milind Girkar, Sergey Kozhukhov, Hideki Saito, Utpal Banerjee, Alexandru Nicolau, Alexander V. Veidenbaum, and Constantine D. Polychronopoulos. On the performance potential of different types of speculative thread-level parallelism: The dl version of this paper includes corrections that were not made available in the printed proceedings. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 24–, New York, NY, USA, 2006. ACM.

[46] K. Kelsey, Tongxin Bai, Chen Ding, and Chengliang Zhang. Fast track: A software system for speculative program optimization. In *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*, pages 157 –168, march 2009.

[47] Hanjun Kim, Arun Raman, Feng Liu, Jae W. Lee, and David I. August. Scalable speculative parallelization on commodity clusters. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 3–14, Washington, DC, USA, 2010. IEEE Computer Society.

[48] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI 2007 Proceedings*, pages 211–222. ACM, 2007.

[49] Shun-Tak Leung and John Zahorjan. Improving the performance of runtime parallelization. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 83–91, New York, NY, USA, 1993. ACM.

[50] Shaoshan Liu, C. Eisenbeis, and J.-L. Gaudiot. Speculative execution on gpu: An exploratory study. In *Parallel Processing (ICPP), 2010 39th International Conference on*, pages 453–461, 2010.

[51] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. Posh: a tls compiler that exploits program structure. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 158–167, New York, NY, USA, 2006. ACM.

[52] Diego R. Llanos. Thread–level speculative parallelization. In P. Alberigo, G. Erbacci, and F. Garofalo, editors, *Science and Supercomputing in Europe, 2004 Annual Report*, pages 211–213. CINECA, Italy, 2005. ISBN 88-86037-15-5.

[53] Diego R. Llanos. Introducción a las técnicas de ejecución especulativa. In *Proceedings of speculative parallelization at running time (UVa)*, October 2008.

[54] Diego R. Llanos. Un modelo software de ejecución especulativa. In *Proceedings of speculative parallelization at running time (UVa)*, October 2008.

[55] Diego R. Llanos, David Orden, and Belén Palop. Meseta: A new scheduling strategy for speculative parallelization of randomized incremental algorithms. In *Proc. 2005 ICPP Workshops (HPSEC-05)*, pages 121–128, Oslo, Norway, June 2005. ISBN 0-7695-2381-1, IEEE Press.

[56] Diego R. Llanos, David Orden, and Belén Palop. New scheduling strategies for randomized incremental algorithms in the context of speculative parallelization. *IEEE Transactions on Computers*, 56(6):839–852, 2007.

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

[58] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 166–176, New York, NY, USA, 2009. ACM.

[59] Ernst P. Mücke, Isaac Saias, and Binhai Zhu. Fast randomized point location without preprocessing in two- and three-dimensional delaunay triangulations. In *IN PROC. 12TH ANNU. ACM SYMPOS. COMPUT. GEOM*, pages 274–283, 1996.

[60] Cosmin E. Oancea, Alan Mycroft, and Tim Harris. A lightweight in-place implementation for software thread-level speculation. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 223–232, New York, NY, USA, 2009. ACM.

[61] Kunle Olukotun, Lance Hammond, and Mark Willey. Improving the performance of speculatively parallel applications on the hydra cmp. In *Proceedings of the 13th international conference on Supercomputing*, ICS '99, pages 21–30, New York, NY, USA, 1999. ACM.

[62] Manohar K. Prabhu and Kunle Olukotun. Exposing speculative thread parallelism in spec2000. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 142–152, New York, NY, USA, 2005. ACM.

[63] Prakash Prabhu, Ganesan Ramalingam, and Kapil Vaswani. Safe programmable speculative parallelism. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 50–61, New York, NY, USA, 2010. ACM.

[64] Joan Puiggali, Boleslaw K Szymanski, Teo Jové, and Jose L Marzo. Dynamic branch speculation in a speculative parallelization architecture for computer clusters. *Concurrency and Computation: Practice and Experience*, 2012.

[65] Carlos García Quiñones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. Mitosis compiler: an infrastructure for speculative

threading based on pre-computation slices. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 269–279, New York, NY, USA, 2005. ACM.

[66] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS XV, pages 65–76, New York, NY, USA, 2010. ACM.

[67] Easwaran Raman, Neil Vahharajani, Ram Rangan, and David I. August. Spice: speculative parallel iteration chunk execution. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 175–184, New York, NY, USA, 2008. ACM.

[68] Ravi Ramaseshan and Frank Mueller. Toward thread-level speculation for coarse-grained parallelism of regular access patterns. In *Workshop on Programmability Issues for Multi-Core Computers*, page 12, 2008.

[69] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160–180, 1999.

[70] Lawrence Rauchwerger and David Padua. The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization. *SIGPLAN Not.*, 30(6):218–232, June 1995.

[71] Peter Rundberg and Per Stenström. Low-Cost Thread-Level Data Dependence Speculation on Multiprocessors. In *Workshop on Scalable Shared Memory Multiprocessors*, June 2000.

[72] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 1–12, New York, NY, USA, 2000. ACM.

[73] Chen Tian, Min Feng, and Rajiv Gupta. Supporting speculative parallelization in the presence of dynamic data structures. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, New York, NY, USA, 2010. ACM.

[74] Chen Tian, Min Feng, Vijay Nagarajan, and Rajiv Gupta. Copy or discard execution model for speculative parallelization on multicores. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 330–341, Washington, DC, USA, 2008. IEEE Computer Society.

[75] Chen Tian, Changhui Lin, Min Feng, and Rajiv Gupta. Enhanced speculative parallelization via incremental recovery. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 189–200, New York, NY, USA, 2011. ACM.

[76] Adrián Tineo, Marcelo Cintra, and Diego R. Llanos. Speculative parallelization of pointer-based applications. In *Poster of the Transactional Access Meeting 2007*, TAM 07, Bologna, Italy, June 2007.

[77] Neil Amar Vachharajani. *Intelligent speculation for pipelined multithreading*. PhD thesis, Princeton, NJ, USA, 2008. AAI3338698.

[78] Christoph von Praun, Luis Ceze, and Calin Caşcaval. Implicit parallelism with ordered transactions. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 79–89, New York, NY, USA, 2007. ACM.

[79] Emo Welzl. Smallest enclosing disks (balls and ellipsoids). In *New results and new trends in computer science*, volume 555 of *Lecture notes in computer science*, pages 359–370. Springer-Verlag, 1991.

[80] Peng Wu, Arun Kejariwal, and Călin Caşcaval. Languages and compilers for parallel computing. chapter Compiler-Driven Dependence Profiling to Guide Program Parallelization, pages 232–248. Springer-Verlag, Berlin, Heidelberg, 2008.

[81] Polychronis Xekalakis, Nikolas Ioannou, and Marcelo Cintra. Combining thread level speculation helper threads and runahead execution. In *ICS 2009 Proceedings*, pages 410–420. ACM, 2009.

[82] Chen Yang and Chu-Cheow Lim. Speculative parallel threading architecture and compilation. In *Proceedings of the 2005 International Conference on Parallel Processing Workshops*, ICPPW '05, pages 285–294, Washington, DC, USA, 2005. IEEE Computer Society.

[83] Paraskevas Yiapanis, Demian Rosas-Ham, Gavin Brown, and Mikel Luján. Optimizing software runtime systems for speculative parallelization. *ACM Trans. Archit. Code Optim.*, 9(4):39:1–39:27, January 2013.

[84] Chao Zhang, Chen Ding, Xiaoming Gu, Kirk Kelsey, Tongxin Bai, and Xiaobing Feng. Continuous speculative program parallelization in software. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 335–336, New York, NY, USA, 2010. ACM.

[85] Chenggang Zhang, Guodong Han, and Cho-Li Wang. Gpu-tls: An efficient runtime for speculative loop parallelization on gpus. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 120–127, 2013.

[86] Zhijia Zhao, Bo Wu, and Xipeng Shen. Speculative parallelization needs rigor: probabilistic analysis for optimal speculation of finite-state machine applications. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 433–434, New York, NY, USA, 2012. ACM.