Universidad de Valladolid

E.T.S. Ingeniería Informática

Trabajo Fin de Grado

Grado en Ingeniería Informática

Desarrollo de un motor de paralelización especulativa con soporte para aritmética de punteros

Autor:
Álvaro Estébanez López
Tutor:
Dr. Diego R. Llanos Ferraris
Dr. Arturo González Escribano

Resumen

La paralelización especulativa es una técnica que permite extraer paralelismo de bucles no analizables en tiempo de compilación. Esta técnica se basa en ejecutar de forma optimista el bucle en paralelo, mientras un sistema monitoriza la ejecución y corrige eventuales violaciones de dependencia. El grupo de investigación Trasgo utilizaba el motor `SpecEngine`, cuyo uso conlleva ciertas limitaciones, como por ejemplo la necesidad de especular sobre un solo tipo de datos y no utilizar aritmética de punteros.

Partiendo de la arquitectura original del motor especulativo, hemos desarrollado una nueva arquitectura que soluciona las principales limitaciones existentes en la versión original. El nuevo motor especulativo permite la especulación sobre datos de diferente tipo en la misma aplicación y aritmética de punteros.

Abstract

Speculative parallelization is a technique that enables to extract parallelism from loops that can not be analyzed at compile time. This technique is based on optimistically executing the loop in parallel, while monitoring system performance and correcting any eventual dependence violation. The Trasgo research group used the engine `SpecEngine`, whose utilization entails certain limitations, such as the need to speculate on a single data type, and the lack of the possibility of using pointer arithmetic.

Based on the original, speculative architecture, we have developed a new architecture that solves the major limitations in the original version. The new speculative engine allows speculation over different data types variables in the same application and uses pointer arithmetic.

Agradecimientos

A lo largo de la realización de mi Trabajo de Fin de Grado 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 Grado, como en el ámbito personal, gracias a él he podido mejorar intentando superarme cada día. Por otro lado, no quisiera olvidarme de Pablo de la Fuente, por la ayuda y los consejos que me proporcionó para el desarrollo de uno de los capítulos del Trabajo.

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 las personas a las que quiero dedicar este Trabajo, mis abuelos Marcelo López y Priscila Alonso, 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.

A Marcelo López y Priscila Alonso

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

La paralelización especulativa es una novedosa técnica desarrollada para los sistemas de memoria compartida que permite ejecutar en paralelo bucles, que a priori, en tiempo de compilación, no eran paralelizables. En este capítulo se proporciona una breve introducción a los fundamentos de la paralelización especulativa. Además se detallan algunos resultados previos logrados por el grupo de investigación Trasgo, cuya principal rama de investigación se basa en la técnica mencionada anteriormente. Por último se facilita al lector un listado de los principales objetivos del proyecto.

1.1. Introducción a la paralelización especulativa

Desde que Intel^{TM} en 1971, sacase a la luz su primer microchip, el 4004[1], ha habido una evolución inmensa. Hoy en día la velocidad de los procesadores está alcanzando unos niveles muy elevados, sin embargo, existe una necesidad implícita de aumentarla. Debido a los avances tecnológicos, empezamos a situarnos en barreras que no se pueden superar, como por ejemplo la velocidad de la luz. Otro inconveniente de los chips, cada vez más sofisticados y repletos de transistores, es el aumento en la disipación de calor, que de alcanzar un nivel muy elevado, podría provocar que el procesador deje de funcionar adecuadamente.

Es por estos motivos por los que empiezan a surgir computadoras con más de un procesador. Existen varias ventajas de los ordenadores multiprocesador respecto a los normales, una de ellas es la posibilidad de disminuir la concentración de calor en un único punto del chip puesto que dividiendo los cálculos entre varios procesadores, el calor también tendrá varios puntos de origen, y no se condensará en un único punto. Otra importante ventaja se puede encontrar en la limitación de velocidad impuesta por la luz, superable si se logra que varios procesadores trabajen concurrentemente para un mismo cometido. Esto hace que actualmente las máquinas tengan integrados varios chips que puedan trabajar al mismo tiempo. Sin embargo, para que la mejora sea sustancial se necesitan programas cuyas instrucciones puedan ejecutarse a la vez, es decir, programas que no se ejecuten en un procesador de forma secuencial. Llevar a cabo este cometido es una ardua tarea, dado que hay que tener en cuenta muchos factores para evitar errores de sincronismo. En la actualidad existen lenguajes

[1] El microchip 4004 tenía una velocidad de 104 KHz y unos 2 300 transistores

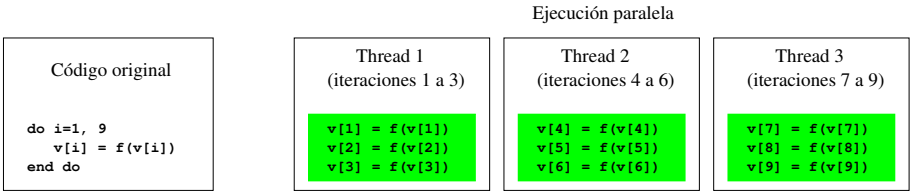

Figura 1.1: Bucle sin dependencias entre los datos de sus iteraciones. Por ser todas las instrucciones independientes, el compilador puede ordenar directamente su ejecución en paralelo.

especializados en estos cometidos, así como extensiones a los lenguajes secuenciales y librerías de funciones. Pero como se puede intuir, se necesitan conocimientos tanto del hardware subyacente, como del problema a paralelizar, así como disponer de las librerías mencionadas. Además crear un software para una determinada arquitectura puede hacer que no sea transportable a otras, creando una indeseable dependencia hacia la máquina. Por tanto lo mejor sería que el compilador se encargase de realizar estas tareas de paralelización. Actualmente existen compiladores que realizan las operaciones descritas anteriormente; sin embargo, la paralelización llevada a cabo por los compiladores actuales, dista mucho de ser óptima, dado que poseen gran recelo a la hora de paralelizar ciertos algoritmos. La razón de ello es la existencia de violaciones de dependencia entre sus datos. Para que un algoritmo pueda ser concurrente las instrucciones que lo componen, no tienen que tener dependencias entre sus datos, cosa que no se puede augurar de un modo sencillo. En la figura 1.1 se puede ver un bucle sin violaciones de dependencia. Por otro lado la figura 1.2 muestra un bucle con posibles violaciones de dependencia.

Existen varias técnicas para abordar la paralelización automática en tiempo de ejecución, por ejemplo el modelo *inspector-ejecutor* y el método de *paralelización especulativa* [3, 4].

El método inspector-ejecutor se resume en la existencia de un bucle inspector extraído del bucle original cuyo propósito no es otro que el de encontrar dependencias cruzadas entre los datos de cada iteración. Cada conjunto de iteraciones que dependen entre sí, son asignadas al mismo procesador para que se ejecuten en orden. Así se obtienen bucles que pueden ser ejecutados en paralelo. Éste método es aconsejable si el procesamiento del bucle inspector es notablemente menor que la carga producida por el bucle original. En muchos casos el cálculo del bucle de inspección proporciona una sobrecarga de procesamiento, a lo que hay que añadir por otro lado la dificultad de realizar un análisis exhaustivo de las dependencias e incluso habiendo casos en que esto es imposible, o bien porque necesita datos de entrada, o utiliza punteros, o utiliza subíndices variables, etc. Es por estos motivos por lo que nosotros centraremos nuestros esfuerzos en el método de *paralelización especulativa*.

1.1.1. Fundamentos de la paralelización especulativa

Este tipo de paralelización se centra en los bucles (como casi todo tipo de paralelización), y considera, de manera optimista, que todos 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 pudieron realizar en tiempo de compilación, suponiendo que no se van a producir violaciones de dependencia entre las iteraciones del bucle. Para realizar su

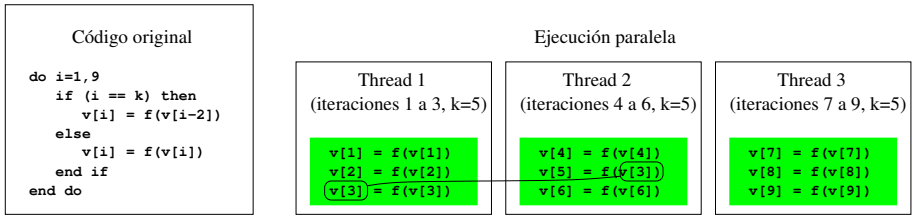

Figura 1.2: Bucle con dependencias entre los datos de sus iteraciones. Dado que el valor `k` no se conoce en tiempo de compilación, el compilador no es capaz de paralelizar el bucle, ya que, si `k=5`, el thread 1 puede que no haya calculado el valor de `v[3]` a tiempo para que el thread 2 lo utilice.

```
for (i=0; i<100; i++)
{
    localVar = sharedVarA+sharedVarB;
    sharedVarA = i*localVar;
}
```

Figura 1.3: Código en lenguaje C que muestra un ejemplo de cada tipo de variable. Así la variable localVar es una variable privada, debido a que primero se escribe sobre ella, para más tarde leer ese valor. En cambio, las variables sharedVarA y sharedVarB son variables compartidas. Como se puede apreciar sharedVarB sólo se lee, por tanto no provocará violaciones de dependencia; por el contrario sharedVarA tanto se lee como se escribe y puede dar lugar a violaciones de dependencia.

objetivo y sin realizar análisis alguno de dependencias, un mecanismo(hardware [2] o software[3], como es nuestro caso) divide el bucle en bloques de iteraciones, a su vez, cada bloque es asignado a diferentes threads de ejecución, destinando cada thread a uno de los procesadores disponibles. Hecho esto, el mecanismo citado supervisa la ejecución de manera que si durante ésta ocurre algún tipo de violación de dependencia, se detiene el proceso y se reinicia con los valores adecuados. Evidentemente, esta tarea de parada y re-ejecución consume tiempo, por lo que cuantas menos violaciones de dependencia se produzcan mejor será nuestra técnica. Por supuesto, cuanto más paralelo sea el bucle, mayores serán los beneficios proporcionados por ésta técnica.

Para entender mejor la paralelización especulativa, es necesario distinguir entre las variables privadas y las compartidas. Se muestra un ejemplo en la figura 1.3.

Las variables privadas son aquellas cuyos valores son modificados en cada iteración, para ser utilizados exclusivamente para dicha iteración, es decir, aquellas cuyo valor se asigna y se lee en la misma iteración y no tienen utilidad más allá de ella. Por el contrario, los valores almacenados en variables compartidas, son aquellos que se utilizan en diferentes iteraciones en cualquier momento de la ejecución. Por tanto, si todas las variables que aparecen en un bucle son privadas, cabe concluir que dicho bucle será paralelizable, y nuestra búsqueda habrá tenido éxito, de modo que el bucle podrá ser ejecutado concurrentemente. Por otro

[2]La solución del mecanismo hardware consiste en módulos hardware adicionales que detectan las dependencias

[3]La solución software consiste en un aumento en el bucle original, creando nuevas instrucciones, que comprueben que no se produzcan violaciones de dependencia durante la ejecución paralela

lado, si las variables son compartidas, no siempre aparecerán violaciones de dependencia. Se producirá una de estas violaciones, si un valor es modificado en una iteración y requerido en una sucesiva, de modo que el valor consultado puede que sea, o no, erróneo. Esto se debe a que no se puede saber si se ha ejecutado de modo adecuado, es decir, del modo que la ejecución secuencial impone. Cuando un thread ejecuta una operación de lectura sobre una variable especulativa, ésta se debe realizar sobre el valor más reciente posible, es decir, por el valor, si existe, asignado por el thread más actual. Todo esto se realiza con el objetivo de conseguir el valor más reciente posible de la variable compartida. En caso de que se produzca una operación de escritura sobre una variable especulativa, se debe comprobar que los demás threads no han sufrido ninguna violación de dependencia, es decir, que ninguno haya obtenido un valor incorrecto de la variable en cuestión.

Existen tres tipos de violaciones de dependencia:

1. **Write-after-write** (WAW): traducido a escritura-tras-escritura, éste tipo de violación de dependencia, se produce cuando se escribe en una variable cuyo valor había sido modificado en una iteración previa. Cuando se produce la situación descrita no se puede asegurar que se vaya a obtener el valor adecuado al finalizar la ejecución. Si bien con palabras es un poco complicado de entender, veamos un ejemplo escrito en código C:

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

   Como vemos hay dos escrituras en la variable *localVar*, una se produce en la iteración 2, y la otra en la 4. En un código secuencial, el valor obtenido al finalizar el bucle en *localVar* sería 7; en cambio, si suponemos que hay dos procesadores, asignamos las iteraciones 1 y 2 al primero, y las iteraciones 3 y 4 al segundo, no podemos saber si se ejecutará antes la iteración 2 que la 4, y por tanto no podremos asegurar que el valor de *localVar* al finalizar el bucle sea 7, pudiendo ser erróneamente 4.

2. **Write-after-read** (WAR): significa escritura-tras-lectura, es un tipo de error que se produce cuando queremos escribir en una variable local cuyo valor se ha leído en una iteración previa, un nuevo valor. Con esto no podemos asegurar que el valor leído es el correcto que se debería obtener. Como en el caso anterior, veamos un ejemplo escrito en código C:

```
// Supongamos que inicialmente
// localVarA == 3
for (i=0;i<4;i++)
{
  if (i==1)
    localVarB = localVarA;
  if (i==3)
    localVarA = 5;
}
```

Se puede contemplar que hay una lectura de la variable *localVarA*, en la iteración 2. También se observa que hay una escritura de esa misma variable en la iteración 4. En un código secuencial, el valor de *localVarA* al finalizar el bucle sería 5, y el valor de *localVarB* sería 3. Pero imaginemos que hay dos procesadores, asignamos las iteraciones 1 y 2 al primero, y las iteraciones 3 y 4 al segundo. No podemos saber si se ejecutará antes la iteración 2 que la 4, y por tanto no podremos asegurar que el valor de *localVarB* al finalizar el bucle sea 3, podría darse el caso en que se ejecutase antes la iteración 4, que la 2, y el valor de dicha variable sea 5 al finalizar.

3. **Read-after-write** (RAW): esta violación de independencia es el caso más peligroso de todos, podemos traducirlo como lectura-tras-escritura. Este error se produce cuando se escribe en una variable compartida un valor que va a ser leído por threads sucesores. Cuando se produce la situación descrita el valor leído por alguno de los threads puede ser incorrecto. Expliquemos este caso con la ayuda de un ejemplo escrito en código C:

```
// Supongamos que sv[1] = 0;
// LocalVar = 1;
for (i=0;i<3;i++)
{
  localVar = sv[1];
  sv[1] = 20;
  if (i==1)
    sv[1] = 10;
}
```

Para ver el error, supongamos que existen tres procesadores, que a cada procesador se le asigna un thread, y que cada thread contiene una iteración, a saber, el thread 1, la iteración 1, el thread 2, la 2 y el thread 3, la 3. En una ejecución secuencial los valores obtenidos serían $localVar = 10$ y $sv[1] = 20$. Pero con nuestra ejecución en paralelo puede que no sea así. Supongamos que en el instante t1 se ejecuta la primera instrucción de los threads 1 y 2, por tanto la versión de la variable *localVar* de cada thread tendrá un valor de 0. Tras esto en el instante t2 se ejecuta la instrucción siguiente del thread 1, quedando sv[1] = 20. Una vez completado, en el instante t3 se ejecuta la primera instrucción del thread 3, dando lugar al error, dado que su versión de *localVar*(la final) será igual a 20 cuando tendría que valer 10. Agregando otro error supongamos que en t4 se ejecuta la instrucción siguiente del thread 3, quedando el valor de sv[1] = 20; y en el instante t5, se ejecuta la última instrucción del thread 2, quedando sv[1] = 10. Al finalizar la ejecución los valores de las variables serán $localVar = 20$ o $localVar = 0$, y $sv[1] = 10$. Todo esto queda resumido en el cuadro 1.1.

Las violaciones de dependencia descritas anteriormente no son irresolubles. Para corregirlas se necesita un control exhaustivo de los accesos a las variables compartidas, de modo que si un thread accede a un valor erróneo sea capaz de desecharlo. Para alcanzar este objetivo cada thread trabaja con su propia versión de variables compartidas, de modo que modifica su versión de la variable compartida, no accede directamente a la versión global.

Una vez finaliza la ejecución de un bloque de iteraciones, ésta puede haber tenido alguna violación de dependencia. Si no hay error alguno, es decir, si no ha habido violaciones de dependencia y la operación ha tenido éxito, se procede a guardar los resultados en la variable

Instante	Hilo 1		Hilo 2		Hilo 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

Cuadro 1.1: Temporización del bucle con los respectivos valores en las variables.

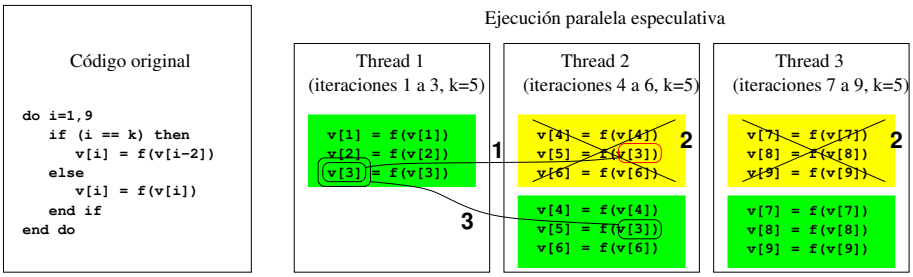

Figura 1.4: La paralelización especulativa inicia la ejecución del bucle en paralelo, mientras un sistema de control monitoriza la ejecución para detectar violaciones de dependencia entre distintos threads. Si se produce una violación, el sistema (1) detiene el thread que ha consumido el valor incorrecto y todos sus sucesores, (2) descarta los resultados generados por éstos, y (3) reinicia los threads de modo que consuman los valores correctos.

compartida global. Esta operación de consolidación de datos se denomina *commit*. Por el contrario, si se produce algún error se descarta la versión de los datos, y se procede a la re-ejecución. Cuando se descarta la versión de los datos se produce lo que se denomina *squash*, y es necesario ejecutar de nuevo todo el bloque de iteraciones asignado al thread desde el principio. Pero no solo se descarta el thread fraudulento en cuestión, sino que además se descartan todos los threads posteriores. Este proceso, denominado *forwarding*, es necesario debido a que cuando un thread va a utilizar el valor de una variable, no sólo consulta su valor local, sino que también tiene en cuenta el valor en los threads anteriores. Por tanto se puede deducir que el valor con que trabaja un thread, si se ha adquirido de un thread que contiene violaciones de dependencia, puede ser erróneo. Se puede ver un ejemplo de esto en la figura 1.4.

1.1.2. Modificaciones en el código original del programa

Los sistemas software de ejecución especulativa deben modificar el código original del programa en tiempo de compilación. Para ello utilizan ciertas funciones que se encargan de:

- *Distribución de bloques de iteraciones.* Las iteraciones tienen que distribuirse entre los threads que participarán en la ejecución especulativa. Para ello pueden utilizarse varias estrategias: distribuir bloques de iteraciones de tamaño constante, adaptarlos a las características de la aplicación, o decidir dinámicamente el tamaño del siguiente bloque a

lanzar.

- *Lecturas y escrituras especulativas.* Como cada thread mantiene copias locales de las variables compartidas, todas las lecturas y escrituras sobre éstas deben reemplazarse por funciones que, además de realizar la operación solicitada, comprueben que no se produzcan violaciones de dependencia.
- *Consolidación de resultados.* Tras la finalización exitosa de la ejecución de un bloque de iteraciones, deberá invocarse a una función que consolide los resultados producidos y solicite un nuevo bloque de iteraciones para su ejecución.

1.1.3. Conclusiones

La paralelización especulativa es una novedosa técnica de paralelización en tiempo de ejecución que permite extraer el paralelismo de códigos secuenciales no analizables en tiempo de compilación. Los resultados obtenidos hasta la fecha muestran que se trata de una técnica válida, aunque su alcance se ve reducido por varias cuestiones que permanecen sin resolver. Estas limitaciones, unidas al hecho de que la paralelización automática no puede aprovechar todo el conocimiento del programador sobre las soluciones inherentemente paralelas, hace que el diseñar e implementar programas directamente paralelos sea imprescindible para aprovechar al máximo las características de estos sistemas.

1.2. Resultados previos del grupo de investigación

El grupo Trasgo tiene como principal hilo de desarrollo, la investigación de técnicas de paralelización especulativa. Así el grupo ha desarrollado un motor que es capaz de realizar las funciones que debe implementar un software dedicado a la paralelización especulativa. En este punto vamos a describir varias de los proyectos llevados a cabo anteriormente por el grupo Trasgo:

- *Toward Efficient and Robust Software Speculative Parallelization on Multiprocessors* [3]. Este artículo contiene un estudio sobre el aumento en la eficiencia que suelen experimentar las aplicaciones que utilizan la técnica de paralelización especulativa. Además cuenta con la descripción del diseño de un software que lleva a cabo la implementación de dicha técnica. Las principales ventajas y novedades del esquema propuesto son que cuenta con un mecanismo de ventana deslizante, una política de sincronización que relaja los requisitos de secciones críticas y estructuras de datos adecuadas. Así prueban que en aplicaciones con pocas violaciones de dependencia, se logran mejoras de hasta el 71 % frente a la versión secuencial. A lo largo del documento se describe en qué consiste la paralelización especulativa, qué son las violaciones de dependencia y qué operaciones debería implementar un software que desarrollase la paralelización especulativa. Hecho esto los autores describen el software que desarrollaron para lograr la deseada paralelización. Por último realizan varias pruebas en varias aplicaciones con pocas violaciones de dependencia, para comprobar la eficiencia del nuevo software, con un amplio estudio de los tiempos, de modo que describen el tiempo empleado en cada una de las operaciones.
- *Design Space Exploration of a Software Speculative Parallelization Scheme* [4]. En este documento nos encontramos una extensión del anterior, pero incluye una aplicación con violaciones de dependencias.

- *New Scheduling Strategies for Randomized Incremental Algorithms in the Context of Speculative Parallelization* [14]. Este artículo se centra en los problemas que producen las violaciones de dependencia en las aplicaciones con un contexto de paralelización especulativa. En concreto analiza algoritmos cuyas violaciones de dependencia tienen menos probabilidad de aparecer cuanto más avanza su ejecución. Se centra en algoritmos incrementales y aleatorios que suelen ser difícilmente paralelizables. Por ello nos introduce en el desarrollo de MESETA [13], una estrategia de programación que tiene en cuenta la probabilidad de aparición de una violación de dependencia para determinar el número de iteraciones a adjudicar.

- *Ejecución paralela de algoritmos incrementales aleatorizados* [9]. Este documento sigue profundizando en el contexto de los algoritmos incrementales aleatorizados, de modo que expone que en el ámbito de la Geometría Computacional, estos algoritmos tienen un claro beneficio en su tiempo de ejecución. Para demostrar esto aborda sendos problemas como son el cálculo de la envolvente convexa y del menor círculo contenedor de nubes de puntos en 2D. Las pruebas realizadas en estas aplicaciones demuestran aceleraciones significativas en la ejecución paralela de bucles frente a su versión secuencial.

- *Paralelización especulativa de un algoritmo para el menor círculo contenedor* [5]. Este trabajo muestra cómo se llevó a cabo la paralelización especulativa de una aplicación. Además muestra los resultados experimentales obtenidos en varias máquinas.

1.3. Objetivos del Trabajo de Fin de Grado

Como se ha introducido al lector, la paralelización especulativa es una técnica de paralelización en tiempo de ejecución que permite la ejecución en paralelo de bucles que no son analizables en tiempo de compilación. Se trata de una técnica experimental que ha demostrado su eficacia en la paralelización de muchos algoritmos de cálculo intensivo. Sin embargo, los motores software de paralelización especulativa no son capaces aún de realizar su tarea en presencia de accesos a estructuras dinámica de datos, es decir, no son capaces de paralelizar bucles que contengan aritmética de punteros.

Este Proyecto busca trabajar en una solución a este problema, modificando el motor de ejecución especulativa desarrollado por Cintra y Llanos [3, 4] para que sea capaz de paralelizar aplicaciones que accedan a datos a través de referencia, no sólo a través del acceso a elementos consecutivos de un vector, como viene realizándose hasta la fecha. Para lograr dicho objetivo seguiremos una serie de etapas:

- *Análisis del problema de la ejecución especulativa de bucles que utilicen aritmética de punteros.* Se llevará a cabo un estudio de la necesidad actual de un software que permita la utilizar punteros en aplicaciones especulativas.

- *Estudio de la arquitectura actual del motor de ejecución especulativa.* Se detallará una descripción del motor actual de paralelización especulativa.

- *Propuesta de modificaciones necesarias para acometer la paralelización objeto de este Trabajo de Fin de Grado.* Se mostrarán las limitaciones actuales, y cómo podríamos solventarlas.

- *Elaboración de varios tipos de soluciones que implementen las modificaciones descritas, y que solucionen las limitaciones actuales.* Se propondrán una serie de arquitecturas del motor especulativo como solución a las restricciones actuales.
- *Elección de la mejor solución.* Se seleccionará una de las soluciones obtenidas, razonando la elección.
- *Implementación del nuevo motor de paralelización especulativa.* Se realizará una implementación de la nueva arquitectura del motor propuesta.
- *Pruebas de rendimiento.* Se darán unos resultados experimentales para lograr obtener unas conclusiones robustas.

A lo largo de los capítulos de este documento, se hará una descripción más detallada de los pasos realizados.

1.4. Alcance del proyecto

El sistema a desarrollar deberá ser capaz de paralelizar especulativamente algoritmos que contengan bucles con aritmética de punteros.

Por soportar la aritmética de punteros, el sistema será capaz de paralelizar los bucles que paralelizaban los motores del pasado, es decir, deberá ser compatible con las aplicaciones que ya utilizaban el motor especulativo original, agregando unas pequeñas modificaciones en el código existente.

1.5. Estructura de este documento

Este proyecto se divide en catorce capítulos, cada uno de ellos tiene como función explicar diferentes cuestiones, así podemos describirlos brevemente del siguiente modo:

- En el primer Capítulo se ha proporcionado una introducción breve a la paralelización especulativa. Además hemos podido conocer diversos trabajos previos del Grupo Trasgo y se han examinado los principales objetivos de este proyecto.
- En el Capítulo 2 procederemos a introducir al lector en el motor especulativo realizado por el Grupo Trasgo en `C`. Para ello se describirán las principales operaciones que realiza, y se mostrará un ejemplo de uso.
- El Capítulo 3 describe las limitaciones presentes en el motor original. Además podemos encontrar las primeras ideas para generar una solución que solvente las restricciones de la arquitectura original.
- En el Capítulo 4 se explica como se ha llevado a cabo la gestión del proyecto. Para ello, se describe el modelo de desarrollo de software utilizado en la elaboración de este proyecto. Además se proporciona la estimación de las horas de trabajo que requerirá la consecución de los objetivos marcados, junto a un análisis de los riesgos que pueden dificultar el desarrollo. Completando lo anterior, se proporcionará un seguimiento de la planificación, para conocer si las previsiones han sido correctas o no. Por otro lado, se hará una descripción de las premisas que se tendrán en cuenta para controlar la calidad, y para la gestión de configuraciones del producto resultante. Por último se calculará el presupuesto necesario para llevar a cabo este Trabajo de Fin de Grado.

- El Capítulo 5 contiene la especificación de los requisitos necesarios para llevar a cabo los objetivos marcados.
- El Capítulo 6 dará una visión de la primera de las soluciones contempladas para crear una nueva arquitectura de ejecución especulativa que solvente las principales limitaciones de la original. Para ello se describirán las modificaciones que habría que llevar a cabo para implementar las operaciones de lectura y escritura especulativas, así como en las operaciones de consolidación. Para finalizar se describe el coste de las operaciones anteriormente mencionadas.
- El Capítulo 7 describe la segunda solución pensada para conseguir un motor especulativo sin las principales restricciones del original. Al igual que en el Capítulo anterior, se describen las modificaciones que deberían realizarse en las operaciones de lectura y escritura especulativas, y en las operaciones de consolidación de datos. También se incluye un análisis de los costes de las operaciones.
- El Capítulo 8 contiene la descripción de la tercera, y última, solución elaborada para conseguir una nueva arquitectura de ejecución especulativa. Este Capítulo sigue el modelo estructural de los dos anteriores, y contiene apartados similares.
- En el Capítulo 9, tras un análisis de las diferentes arquitecturas propuestas, escogeremos una de ellas, dando las razones de dicha elección.
- El Capítulo 10 muestra cómo se ha llevado a cabo la implementación de la nueva arquitectura del motor, comentando las estructuras de datos que poseerá, y la implementación de las principales operaciones a desarrollar.
- El Capítulo 11 contempla las pruebas que se han llevado a cabo para comprobar la validez de la implementación realizada.
- El Capítulo 12 describe los bechmarks que se van a utilizar para la obtención de resultados experimentales.
- El Capítulo 13 contiene los resultados experimentales de la ejecución de los benchmarks, comparando la arquitectura del motor original, con la del nuevo motor especulativo.
- Por último en el Capítulo 14 se detallarán las conclusiones que se han podido extraer de nuestro trabajo, así como una exposición de posibles vías de continuación al Trabajo de Fin de Grado realizado.
- En el Apéndice A se propiciará una descripción de los contenidos del CD-ROM adjunto a la memoria.
- En cuanto al código de la nueva arquitectura de paralelización especulativa, así como los bechmarks desarrollados, se encuentran en el CD-ROM adjunto a la memoria.

Capítulo 2

Descripción del motor especulativo original

Con el objetivo de clarificar la visión acerca del motor de paralelización especulativa, se facilita una introducción y una descripción de su funcionamiento. Para ello explicaremos las operaciones que efectúa, así como las estructuras de datos con las que se implementa.

2.1. Introducción

Los motores software de ejecución especulativa buscan ejecutar bucles en paralelo donde la dependencia entre sus datos es desconocida. Este motor [11] se basa en un conjunto de librerías de funciones que permiten ejecutar especulativamente un bucle [3, 4, 10]. Es un sistema en construcción, dado que actualmente es el programador el encargado de agregar el código adicional necesario para lograr la paralelización. En un futuro se espera lograr que dicho proceso sea automático.

Procedamos a ver un ejemplo de cómo se ejecutaría especulativamente un bucle. Para ello nos basaremos en el bucle de la figura 2.1. Este bucle consta de cuatro repeticiones, supongamos que disponemos de varios procesadores y que cada iteración del bucle se ejecuta en uno diferente.

Ahora tenemos cuatro threads, el correspondiente a la iteración más baja puede conside-

```
do i=1, 4
   ...
   LocalVar1 = SV[x]
   SV[x] = LocalVar2
   ...
end do
```

Figura 2.1: Ejemplo de bucle con dependencia RAW. Este tipo de bucle puede dar lugar a violaciones de dependencia en su ejecución paralela.

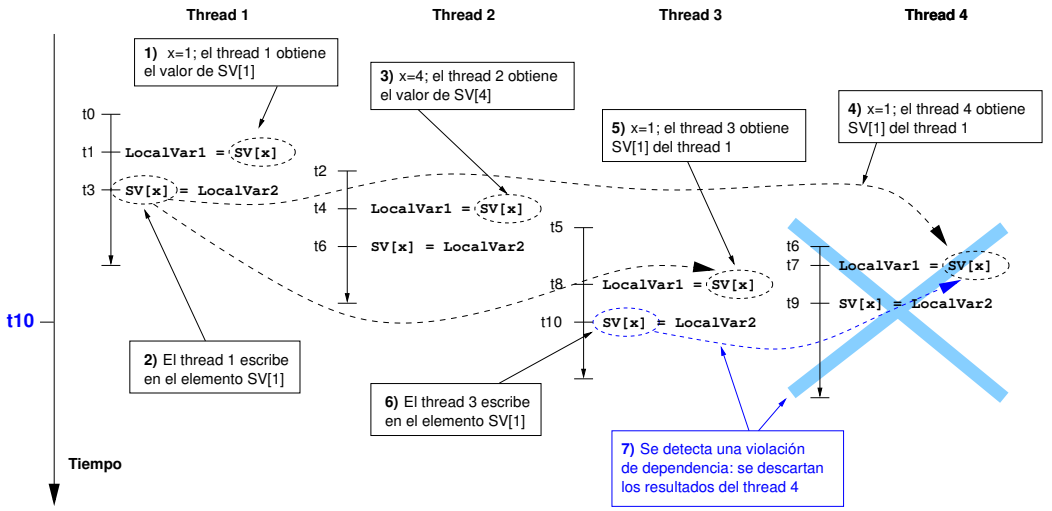

Figura 2.2: En cada lectura se busca la versión más actualizada de `SV[X]`, consultando a los threads anteriores hasta llegar, en su caso, al thread no especulativo. En cada escritura se comprueba si un thread posterior ha utilizado un valor incorrecto, en cuyo caso se descarta su ejecución.

rarse como thread no especulativo, y el que corresponde a la última, será considerado como thread más especulativo (estos conceptos se abordarán con más detalle a lo largo de este capítulo). Como sabemos del capítulo anterior, cada thread mantiene su versión de los datos compartidos, por tanto pueden aparecer incoherencias al finalizar el bucle, si cada uno consolida sus resultados de forma aleatoria, es por eso por lo que cada thread debe consolidar sus datos de modo ordenado. Para comprender mejor los datos expuestos mostraremos un ejemplo de ejecución en la figura 2.2. En esta figura se puede apreciar una posible ejecución en paralelo del bucle de ejemplo mostrado en la figura 2.1. Se puede ver que todas las operaciones se producen de un modo que no induce errores, hasta que llega el instante **t10**. En este momento el thread tres modifica el valor del vector compartido utilizado en el instante **t7** por el thread cuatro, provocando que se deban descartar los resultados obtenidos desde ese momento por el thread cuatro.

Para lograr evitar fallos indeseables, se necesita hacer una serie de modificaciones en el código original. Estas modificaciones se pueden ver en la figura 2.3, y son las siguientes:

1. Las **operaciones de lectura** sobre variables especulativas se sustituyen por una función que recupera el valor más reciente, es decir, el más actualizado, de la variable.

2. Las **operaciones de escritura** sobre variables especulativas se reemplazan por una función que realiza la escritura y detecta violaciones de dependencia esporádicas.

3. Cada thread ejecuta al final de la ejecución de su bloque, una **operación de consolidación de los datos calculados** y asigna el nuevo bloque de iteraciones a ejecutar.

En este punto cabe mencionar que para poder llevar a cabo la mencionada paralelización, deben tenerse en cuenta una serie de **requisitos**. Como pre-requisitos para realizar su

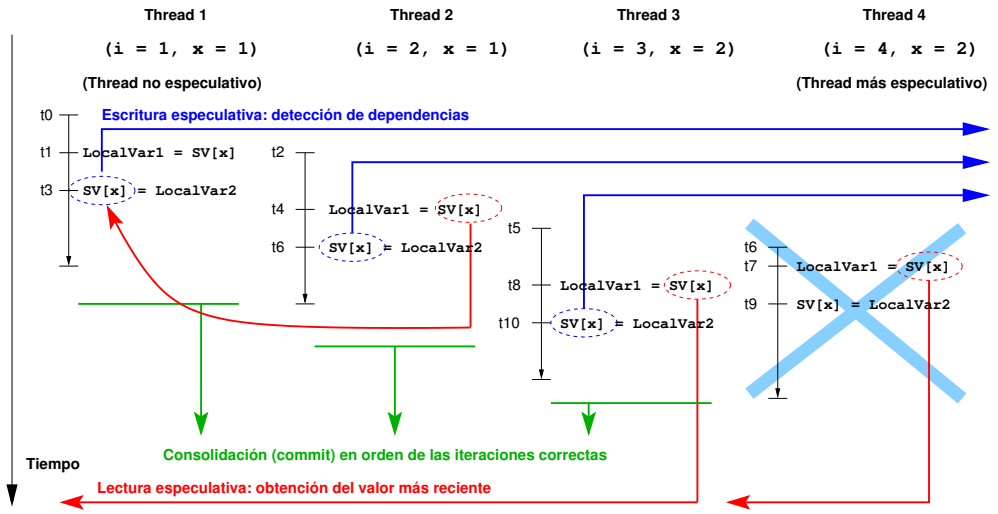

Figura 2.3: La escritura especulativa realiza las revisiones de datos recorriendo los threads posteriores, es decir, va de menos especulativo a más especulativo. Por el contrario, la lectura especulativa busca el dato solicitado recorriendo los threads anteriores, es decir va hacia los threads menos especulativos. La operación de consolidación se realiza en orden creciente de los threads.

cometido, se pueden enumerar los siguientes:

1. El bucle a analizar debe tener un **número de iteraciones conocido** antes de la ejecución.
2. El bucle que queremos paralelizar **no ha de trabajar con aritmética de punteros**.
3. Dentro del bucle a paralelizar **no se podrá utilizar memoria dinámica**.

Una vez aseguradas estas premisas, el motor realiza un **análisis previo** a la ejecución del bucle. En dicho estudio clasifica las variables disponibles, así etiqueta como **privadas** las variables sobre las que siempre se escribe, antes de utilizarlas dentro de una misma iteración. Como variables **compartidas** se etiquetan aquellas que simplemente son leídas. El resto de las variables se etiquetan como **especulativas**, y son las que son susceptibles de sufrir violaciones de dependencia.

En este punto se debe señalar que además de los requisitos que debe cumplir el bucle, de la modificación que sufren las operaciones mencionadas, y de la clasificación de las variables, se deben realizar **modificaciones en la estructura del bucle** del siguiente modo: supongamos que tenemos un bucle que cuenta con N iteraciones y se desea paralelizar de modo que la ejecución la llevarán a cabo P procesadores. Para alcanzar dicho objetivo se sustituye el bucle original por un *doall* de *1* a P, y se asigna a cada procesador un bloque de iteraciones consecutivas. Cuando un thread termina su bloque, invoca a una función encargada de asignarle un nuevo bloque. Cuando no queden bloques que asignar, y todos los threads finalicen, se da por terminada la ejecución.

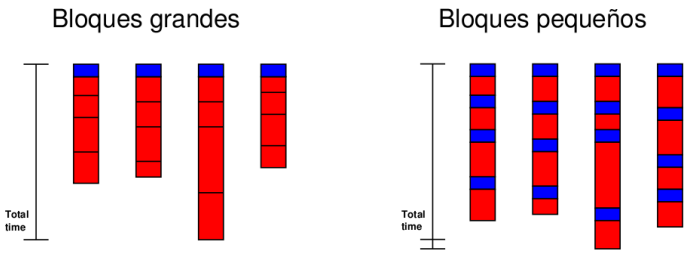

Figura 2.4: Comparación de tamaños de bloques.

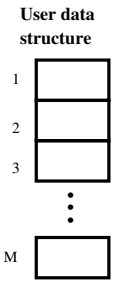

Figura 2.5: Estructura de datos necesaria cuando el número de variables especulativas sea `M`.

Como última observación, se puede considerar que para optimizar el tiempo, conviene que los **tamaños de los bloques** de iteraciones sean **pequeños**, aún gastando más tiempo en la asignación. Si no fuese así podría haber bloques muy grandes que retrasasen toda la ejecución. Véase Figura 2.4.

2.2. Funcionamiento del motor especulativo

El motor especulativo trata de evitar violaciones de dependencia, para lo cual utiliza ciertas estructuras de datos.

2.2.1. Estructuras de datos auxiliares

La primera estructura de datos sobre la que habría que hacer hincapié son los vectores. Estas estructuras de datos son las encargadas de constatar las operaciones a las que están sometidas las variables especulativas en cada operación. Por tanto esta estructura de datos tendrá un tamaño igual al número de variables especulativas existentes en el bucle, un ejemplo en la figura 2.5. Cabe señalar que cada thread contiene su propia versión de los datos especulativos, luego como se verá a continuación, deberá haber al menos tantos vectores de este tipo, como procesadores.

El motor cuenta con otra estructura, otro vector. Este vector será el encargado de administrar los procesadores asignándoles threads, así será una ventana deslizante (*sliding window*).

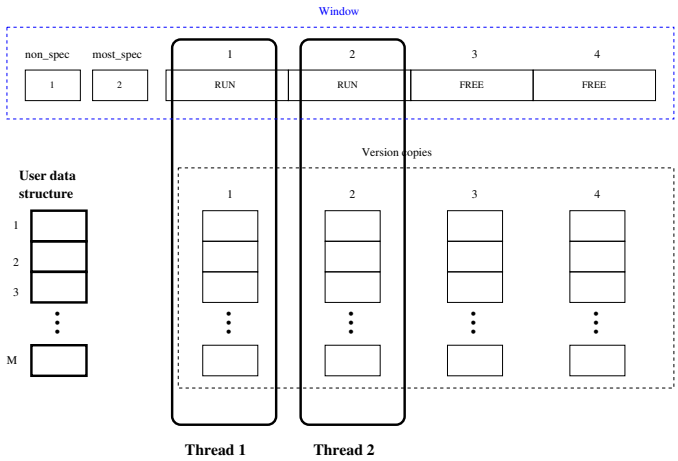

Figura 2.6: Motor con las estructuras de datos descritas en la sección 2.2.1.

El número de slots o huecos, es decir, el tamaño de este vector, debe ser mayor o igual que el número de procesadores disponibles para que todo funcione correctamente. Luego si suponemos que hay `W` huecos y `P` procesadores, $W \geq P$. Para cada hueco, se dispone de una versión de los datos, además de una global, es decir habrá `W+1` estructuras con `M` (nºde variables especulativas) elementos cada una. Cada hueco de la ventana tiene asignado un estado que marca la situación actual del slot. Los estados pueden tomar los siguientes valores:

- **FREE**: el slot está libre y puede ser asignado a un thread. Inicialmente todos los huecos de la ventana deslizante toman este valor.
- **RUNNING**: el slot está ocupado por un thread en ejecución.
- **PENDING_SQUASH**: este estado viene provocado por la detección de que el thread está utilizando un valor incorrecto en una de sus variables especulativas, y por tanto se tiene que invalidar la ejecución de dicho thread.
- **SQUASHED**: el slot está ocupado por un thread que debe comenzar de nuevo su ejecución, porque se ha detectado una violación de dependencia.
- **DONE**: este estado indica que el thread que ocupa este hueco ha finalizado su ejecución con éxito.

La ventana permite la asignación consecutiva de threads hasta completar los `P` disponibles en ese momento, y estos `P` huecos pasan a estar en ejecución (estado RUN). A medida que cada thread realiza sus operaciones y finaliza, la ventana de trabajo se desplaza a la derecha, por tanto la ventana además de disponer de `W` huecos, contendrá dos apuntadores, uno al primer hueco en uso, también denominado no especulativo, y otro al último, el más especulativo. Hasta ahora tenemos el conjunto de estructuras de datos descritas en la figura 2.6.

Hemos visto que cada hueco se asigna a un procesador, y tiene asociada una versión (estructura de datos) de las variables especulativas. Además de conocer el valor de los datos

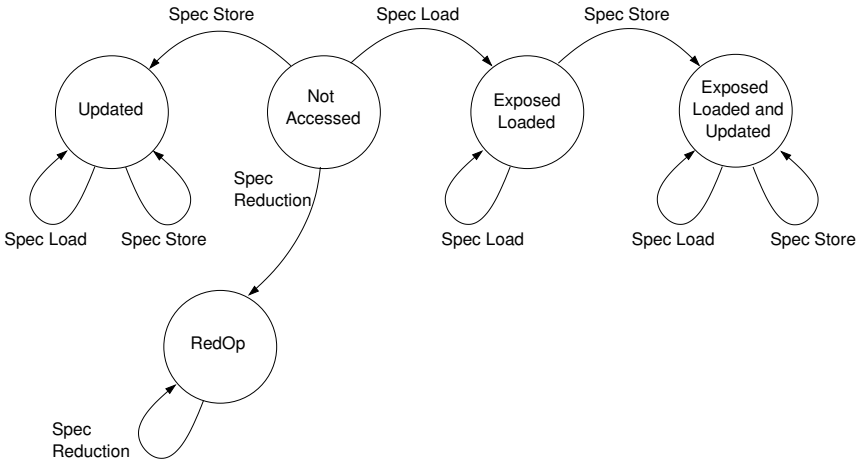

Figura 2.7: Evolución de los estados posibles de cada elemento de la matriz de acceso.

propio a su thread, se debe conocer el estado de cada variable problemática, es decir, si ésta ha sido leída, modificada, etc. Es por esto que cada hueco también dispondrá de una **matriz de acceso** con `M` posiciones. Los elementos de esta matriz contendrán el estado de la versión de cada variable. Estos estados pueden ser los siguientes:

1. **NotAcc**: elemento no accedido aún por dicho thread, éste es el estado inicial de cada elemento.
2. **ExpLd**: *Exposed Loaded*, quiere decir que el thread ha leído el valor de la variable de ese elemento.
3. **Update**: el thread ha modificado el valor de la variable que implementa ese elemento.
4. **ElUp**: este estado quiere decir que el thread ha consultado el valor de una variable para más tarde modificarlo.
5. **RedAdd**: se ha aplicado una operación de reducción sobre la suma con este elemento.
6. **RedMax**: se ha aplicado una operación de reducción sobre el máximo con este elemento.

En la figura 2.7 se puede ver una especie de autómata cuyos estados son los que soporta la matriz de acceso y que evolucionan a través de operaciones de lectura o escritura. En la figura 2.8 se puede ver el estado del motor especulativo con las estructuras vistas hasta el momento.

Una vez vistos los diferentes estados de los componentes de la matriz de acceso procedamos a describir las distintas operaciones que lleva a cabo el motor especulativo.

2.2.2. Lectura especulativa

En tiempo de compilación cada vez que una variable va a ser leída, se sustituye por una llamada a función, del siguiente modo:

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

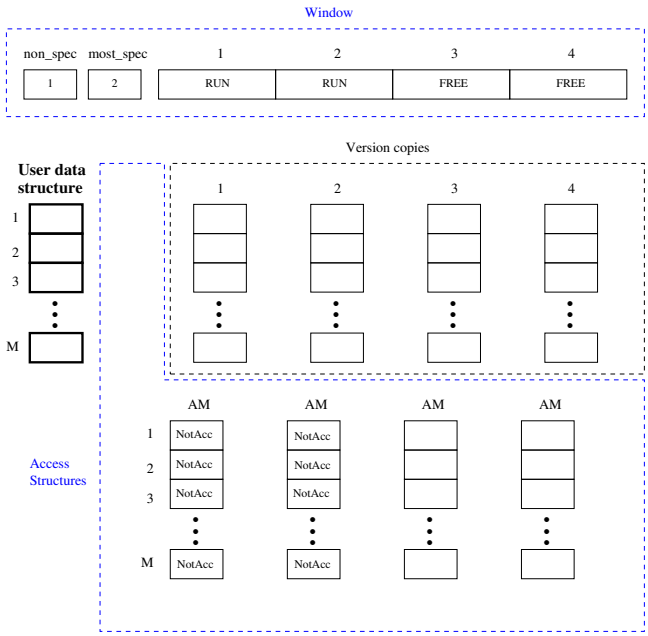

Figura 2.8: Estructura del motor especulativo original.

⇓

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

Los argumentos tienen el siguiente significado:

1. **index**: posición del elemento en el vector especulativo.
2. **current**: número de slot asignado.
3. **LocalVar**: variable donde se almacenará el dato consultado.
4. **maincopy**: estructura de datos compartida de la que se lee el elemento de la posición *index*.
5. **myIVtail**: último elemento ocupado del vector de indirección (estructura descrita en la sección de consolidación de resultados de este capítulo).

Supongamos que un thread cualquiera `N` quiere leer un valor del vector especulativo. Veamos como se leería la variable:

1. Consulta el valor de la posición de la variable en su matriz de acceso para obtener el valor más actualizado posible. Si es *NotAcc* el thread no dispone de una copia, y recurre a sus predecesores.
2. Busca entre los `N-1` anteriores threads el valor de la variable.

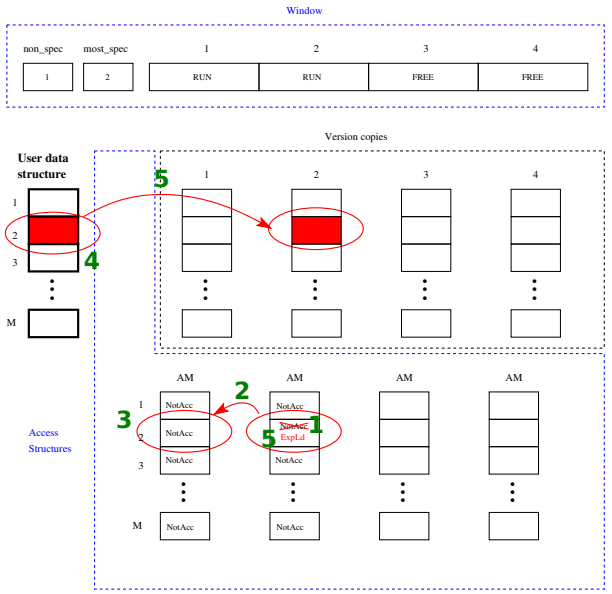

Figura 2.9: Ejemplo de operación de lectura.

3. En caso de que ninguno haya utilizado el valor deseado, el thread `N` recurrirá al valor de referencia y almacenará dicho valor en su versión de los datos. En caso de que alguno de los threads anteriores poseyese el dato lo leería de su versión (clara necesidad del *forwarding*).

4. Hecho esto actualiza el valor del elemento en su matriz de acceso al estado *ExpLd*.

A continuación veremos las operaciones recién descritas aplicadas a un ejemplo concreto, para ello nos basaremos en la figura 2.9.

1. Supongamos que el thread 2 debe ejecutar la instrucción LocalVar=SV[2].

2. El thread 2 necesita el elemento 2, pero no dispone de copia: buscará una entre sus predecesores.

3. El thread 1 (predecesor del thread 2) no ha utilizado el dato.

4. Ningún predecesor ha utilizado este dato: el thread 2 recurrirá al valor de referencia.

5. El valor de referencia se almacena en la versión correspondiente al thread 2 y se actualiza el estado de ese elemento a ExpLd (Exposed Loaded), completándose la instrucción LocalVar=SV[2].

Early Squashing

En las operaciones de lectura que ejecuta el motor existe una opción que puede mejorar el rendimiento de la aplicación. Esta opción se denomina *Early Squashing* y consiste en ejecutar una operación de squash antes de que el bloque de iteraciones finalice su ejecución. Para comprender su funcionamiento, supongamos que nuestra aplicación realiza un *specstore* (operación de escritura descrita en el apartado 2.2.3), esta operación comprueba los slots de los threads posteriores. Si en esta comprobación detecta una violación, pone en SQUASH estos slots, pero su ejecución continúa, hasta finalizar. Por el contrario, si introducimos la operación *Early Squashing* y durante la ejecución de un slot hay una operación de lectura (*specload*), dicho thread comprueba que su estado no sea SQUASH, en cuyo caso, retorna un valor negativo en el parámetro *LocalVar* y finaliza la ejecución de *specload*. Con esta forma conseguimos perder menos tiempo ejecutando instrucciones innecesarias, dado que como el estado es SQUASH, los resultados de la ejecución serán desechados. Por tanto cuando realizamos una operación de lectura en nuestra aplicación comprobaremos el valor de *LocalVar*, de modo que si esta variable tiene un valor igual a `-1`, la operación de *Early Squashing* provocará un "salto" hasta la parte del código donde el thread finaliza la ejecución de su bloque.

2.2.3. Escritura especulativa

En cuanto a las operaciones de escritura, también se sustituyen en tiempo de compilación por una llamada a función de modo parecido al descrito en las operaciones de lectura:

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

⇓

specstore(index, current, LocalVar, myIVtail)

Los argumentos tienen el siguiente significado:

1. **index**: posición del elemento a modificar en el vector especulativo.
2. **current**: número de slot asignado.
3. **LocalVar**: variable que contiene el dato que se desea guardar en el vector especulativo.
4. **myIVtail**: último elemento ocupado del vector de indirección (estructura descrita en la sección de consolidación de resultados de este capítulo).

Supongamos que un thread cualquiera `N` quiere escribir una variable especulativa. Veamos los pasos necesarios para realizar dicho cometido:

1. El thread que realiza la operación, modifica el valor de la posición *index* de su propia versión.
2. Modifica el estado de la posición *index* en su matriz de acceso, así si su estado anterior era *Not Accessed*, pasaría a *Updated*. Si su estado era *Exposed Loaded* pasaría a ser *Exposed Loaded and Updated (ElUp)*.
3. Comprueba si algún thread sucesor ha utilizado alguna versión antigua de los datos, consultando sus matrices de acceso.
   a) Si los threads posteriores no habían utilizado este dato, véase figura 2.10, la operación de escritura concluye.

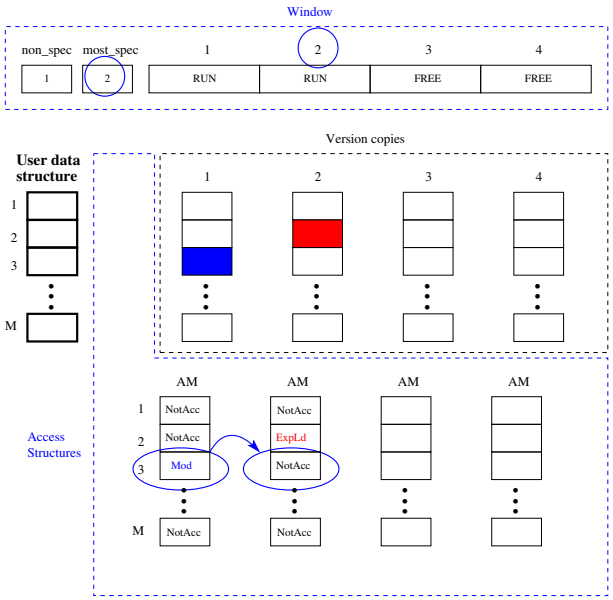

Figura 2.10: Descripción de una operación de escritura en que los threads posteriores no han utilizado el dato a modificar.

b) Si los threads posteriores habían utilizado una versión antigua, y por lo tanto fraudulenta, del dato, véase figura 2.11:
   1) Se detiene la ejecución del thread que utilizó ese dato incorrecto y la de todos sus sucesores (*forwarding*).
   2) El estado de dichos threads pasa de RUN a SQUASHED.
   3) Se reinicia la ejecución de todos los threads con estado SQUASHED, pero con los valores actualizados en sus versiones.

Para afianzar la operación descrita, vamos a exponer un ejemplo concreto, basándonos en la figura 2.12.

1. Supongamos que el thread 1 ejecuta SV[2]=LocalVar: en primer lugar se realiza la escritura.
2. El estado del elemento era NotAcc, por lo que pasa a Mod (si fuera ExpLd pasaría a ExpLdMod).
3. A continuación se comprueba si algún thread sucesor ha utilizado una versión antigua de este dato: éste es el caso.
4. Se detiene la ejecución del thread que ha consumido el dato incorrecto y de todos sus sucesores.
5. La ejecución se reiniciará en el thread 2 desde el principio, de modo que la lectura especulativa cargue el valor correcto.

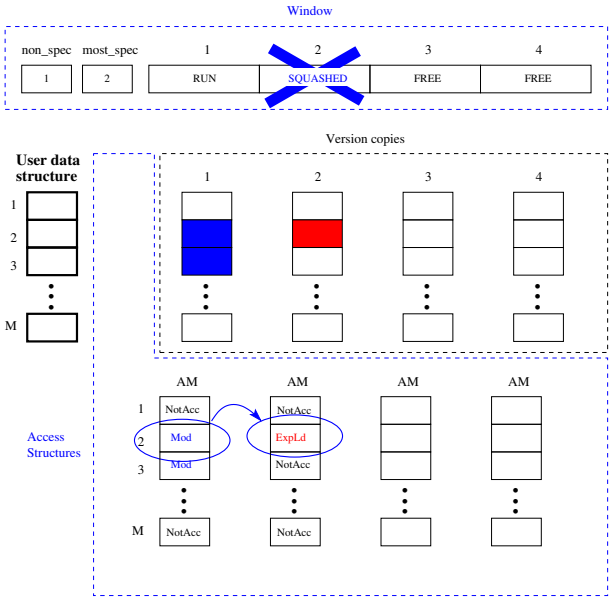

Figura 2.11: Descripción de una operación de escritura en que los threads posteriores han utilizado el dato a modificar, y por tanto deben reiniciarse.

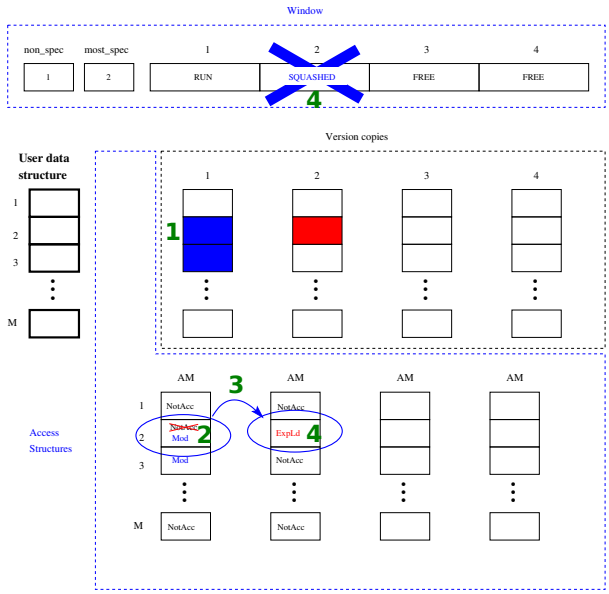

Figura 2.12: Ejemplo de operación de escritura.

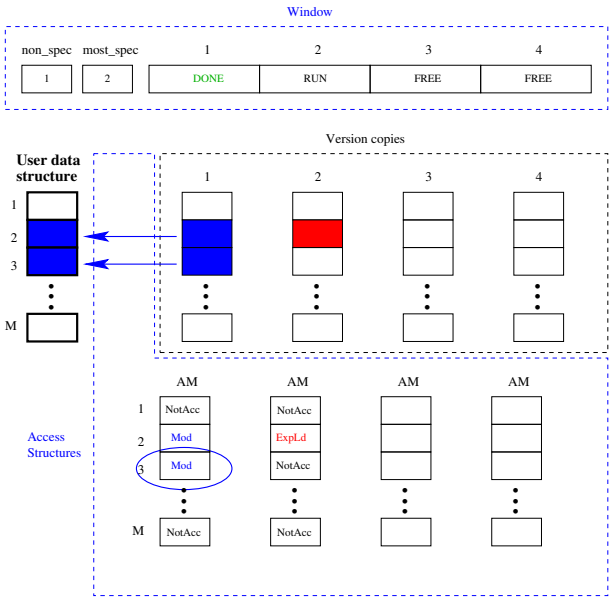

Figura 2.13: Descripción de una operación de commit. Sólo se vuelcan a la versión global los valores modificados en la versión local.

Cabe señalar que las operaciones de escritura afectan a las versiones locales de los threads, de modo que sólo afectarán a la versión global cuando se produzca la operación de consolidación de datos.

2.2.4. Consolidación de resultados

Una vez finalizada la ejecución de un bloque de iteraciones, el estado del thread pasa a DONE y se procede a invocar una función encargada de realizar las operaciones de consolidación (commit). La tarea de esta operación es transmitir los valores de la copia local del thread, a la copia principal. Además de esto, la función se encarga de asignar el siguiente bloque de iteraciones a ejecutar.

Para realizar la operación de *commit* cuando termina un thread, vuelca los valores modificados en su versión, véase figura 2.13, a la estructura de datos principal. Una vez hecho esto se pone el slot en estado FREE (libre). Cabe señalar que si es el primer thread se debe aumentar el apuntador *non-spec*. Por último se le asigna al thread que ha finalizado su bloque, el siguiente slot libre con un nuevo bloque de iteraciones, y se aumenta el apuntador *most-spec*. Se puede comprobar ahora como ese thread pasa de ser el no especulativo, a ser el más especulativo.

La función que implementa esta operación será descrita en el apartado de optimización de la consolidación de resultados.

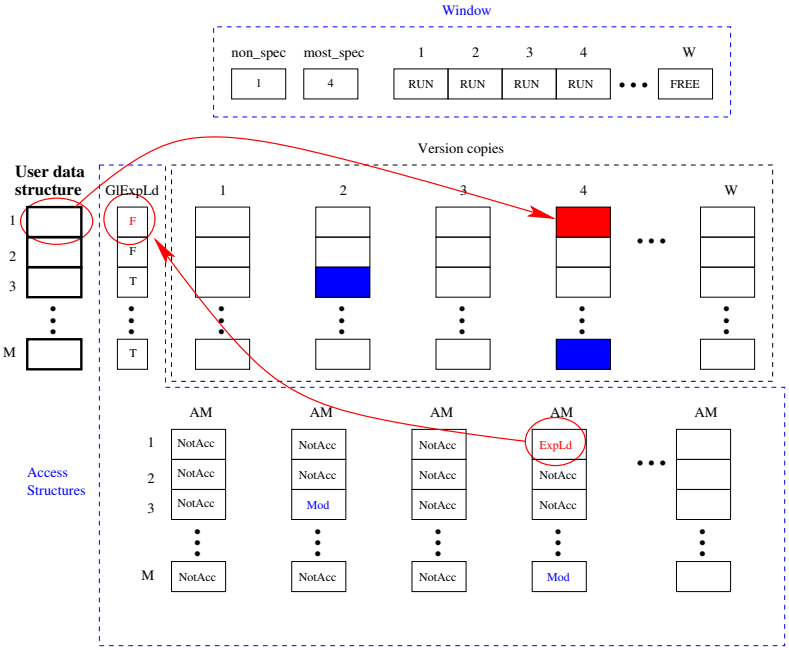

Figura 2.14: Vista del funcionamiento del vector *Global Exposed Load*.

2.2.5. Optimizaciones

A las operaciones anteriormente expuestas, se les puede realizar un conjunto de mejoras que procederemos a exponer, a saber, una optimización de la lectura especulativa y una optimización del *commit*.

Optimización de la lectura especulativa

Para realizar una lectura, siguiendo lo antes expuesto, un thread recorre las versiones de todos los threads predecesores buscando el valor actualizado a consultar. Sin embargo, normalmente los datos que solicita un thread no han sido utilizados, es decir, el thread recurre al valor de referencia para completar una operación de lectura. Para evitar la lectura inútil de todas las versiones de los threads anteriores, se incorpora un vector auxiliar denominado *Global Exposed Load*. Dicho vector indica si citada variable ha sido utilizado en alguna ocasión, por tanto hay que actualizarlo cada vez que se produzca una operación de lectura o escritura en los elementos de cada versión de los datos. Véase Figura 2.14.

Optimización del *commit*

La operación de consolidación de resultados requiere la comprobación de todos los elementos de la versión del thread, lo que implica comprobar tanto los que han sido modificados, como los que no. Una mejora de dicha operación puede conseguirse implementando una lista de elementos modificados, denominada *Indirection Matrix*. Por tanto cada thread dispondrá de su propio vector de elementos modificados, con la posición del elemento utilizado. Además

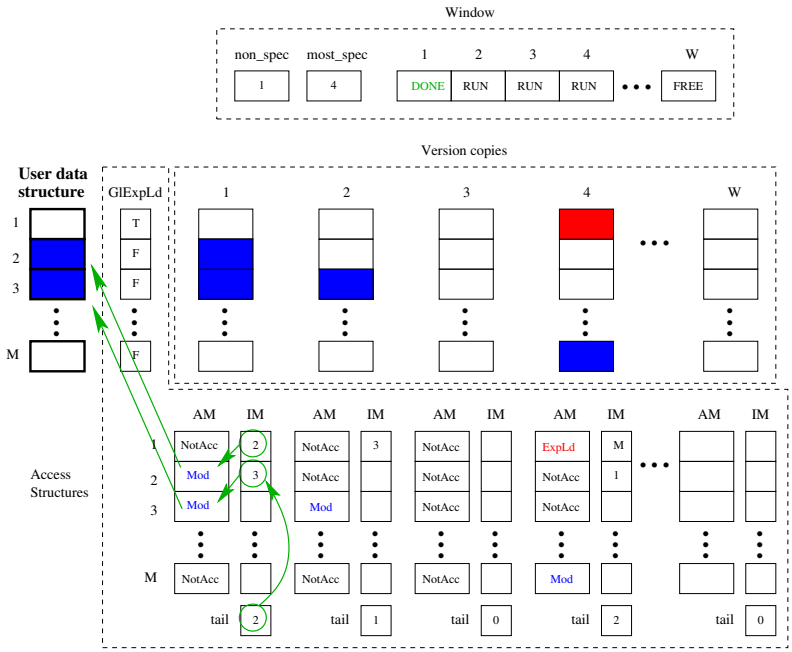

Figura 2.15: Vista del funcionamiento del vector*Indirection Matrix*.

cada thread contará con un campo llamado *tail* cuya función será indicar la posición del último elemento de la lista *Indirection Matrix*, para facilitar el recorrido de ésta. Ahora para realizar la operación de *commit*, basta con volcar los elementos de la lista. En la figura 2.15 podemos ver la aplicación de esta estructura.

La función que permite implementar la operación de consolidación, se denomina **threadend** y está formada por los siguientes argumentos:

threadend(current, retflag, maincopy, myIVtail)

Con el siguiente significado:

1. **current**: número de slot asignado.
2. **retflag**: valor de retorno de la función. Se utiliza para saber si quedan más bloques por asignar. Puede devolver dos valores, a saber, `JOBTODO` o `JOBDONE`. Si retorna `JOBTODO`, todavía hay bloques de iteraciones por ejecutar. En cambio si el valor es `JOBDONE`, los bloques de iteraciones que forman el bucle han sido asignados al completo.
3. **maincopy**: estructura de datos compartida en la que se consolidarán los datos.
4. **myIVtail**: último elemento ocupado de la matriz de indirección.

Podemos apreciar el resultado final del motor, con todas las estructuras implementadas en la figura 2.16.

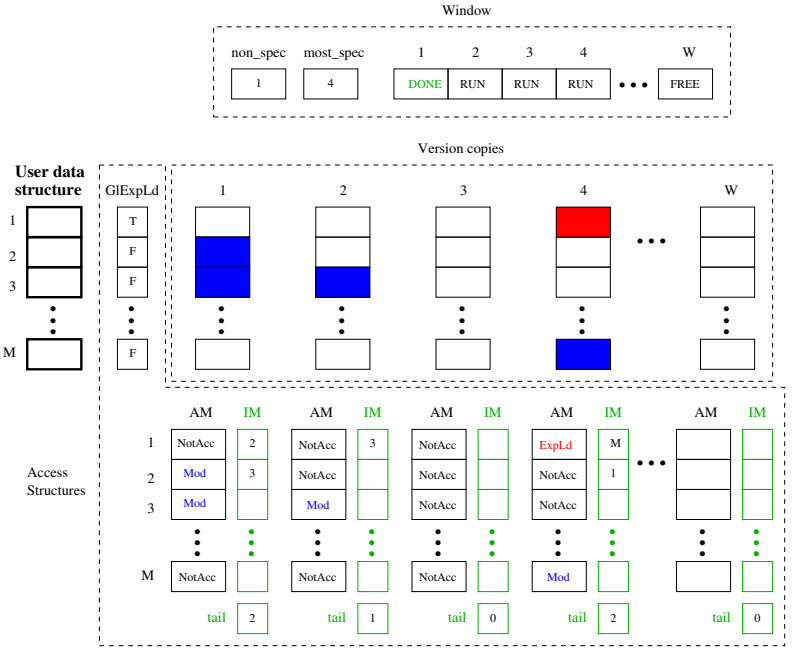

Figura 2.16: Motor especulativo al completo, con todas sus estructuras auxiliares.

2.2.6. Operaciones de reducción

Existen ciertas operaciones que no pueden ser paralelizables por la naturaleza de la instrucción. Sin embargo, dentro de estas situaciones, hay cierto tipo de operaciones a las que se puede aplicar una transformación con el fin de paralelizarlas. En estos casos se dice que la operación se puede *reducir*. Por ejemplo podemos reducir las operaciones de suma y de cálculo del máximo.

Reducción de la suma

Imaginemos que en el bucle a paralelizar existe una operación de suma del siguiente modo:

$$\textbf{matriz(i)} = \textbf{matriz(i)} + \textbf{valor}$$

Esta operación produciría en el bucle una operación de suma por cada iteración. Como se puede intuir, esta operación de suma modificará el valor del elemento `i` de la `matriz`, de modo que si nos propusiésemos paralelizarlo, provocaría continuas violaciones de dependencia. Esto es debido a que la iteración que modificase el valor de `matriz(i)` necesitaría el valor que la iteración anterior le proporcionase, es decir, la iteración `J`, necesita conocer el valor de `matriz(i)`, en la iteración `J-1` para actualizar el valor.

Para solucionar este problema se realiza una operación de reducción a la suma de la siguiente manera:

$$\textbf{matriz(i)} = \textbf{matriz(i)} + \textbf{valor1} + \textbf{valor2} + \textbf{...} + \textbf{valorK}$$

Así en vez de sumar una cantidad en cada iteración, se suman todas a la vez permitiendo evitar violaciones de dependencia. En el motor esta operación se implementa mediante la función **specadd**:

```
matriz(i) = matriz(i) + valor
```
$$\Downarrow$$
specadd(index, current, value, myIVtail)

Dónde los argumentos significan:

1. **index**: posición del elemento dentro del vector especulativo dónde se realizará la suma.
2. **current**: número de slot asignado.
3. **value**: la cantidad que se desea sumar en cada iteración.
4. **myIVtail**: último elemento ocupado del vector de indirección.

El funcionamiento de esta operación modifica la forma de actuar de la operación de consolidación. Para ello cada thread va acumulando en su versión de los datos la cantidad que suma en cada iteración, de modo que si tiene `N` iteraciones, al finalizarlas el valor que contendrá será `N*valor`. En ese momento la operación de consolidación, en lugar de modificar el valor de la estructura global con su valor `N*valor`, lo que hace es sumárselo, consiguiendo los resultados deseados al final de la ejecución.

Reducción del máximo

Imaginemos que en el bucle a paralelizar aparece una operación de cálculo del máximo del siguiente modo:

SI **matriz(i)** $<$ *valor* ENTONCES
matriz(i) $=$ *valor*

o directamente así:

matriz(i) = MAX(matriz(i),valor)

Esta operación no puede paralelizarse debido a que, como en el ejemplo anterior, produciría continuas violaciones de dependencia. Pero también como en el caso de la suma vista, se puede *reducir* esta operación. Con ello se obtiene lo siguiente:

matriz(i) = MAX(matriz(i),valor1,valor2,...,valorK)

Inicialmente se calculaba el máximo entre los dos valores posibles, sin embargo, ahora se calcula sobre todas las cantidades posible, permitiendo la paralelización. La función que implementa esta operación en el motor es la siguiente:

```
matriz(i) = MAX(matriz(i),valor)
```
$$\Downarrow$$
specmax(index, current, value, myIVtail)

Con el siguiente significado de los argumentos:

1. **index**: posición del elemento dentro del vector especulativo dónde finalmente quedará el valor máximo.
2. **current**: número de slot asignado.
3. **value**: es la cantidad sobre la que se calcula el máximo.
4. **myIVtail**: último elemento ocupado del vector de indirección.

Esta operación se comporta de la siguiente manera: cada thread calcula el máximo de los valores que corresponden a su bloque de iteraciones. Cuando finaliza su ejecución y procede a consolidar los datos, compara ese valor máximo con el del vector que contiene la copia global de los datos. Si el valor es mayor, lo deposita en la posición correspondiente; sino no hace nada.

2.2.7. Funciones de inicialización del motor

Existe un paso previo a la implantación de instrucciones paralelas en un código. Así es imprescindible inicializar las estructuras de datos de que consta el motor antes de realizar cualquier operación. Para realizar estas inicializaciones se utilizan dos funciones, a saber, **specinit** y **specstart**. El objetivo de *specinit* es principalmente controlar el tamaño de los bloques que se fijará a los slots, este tamaño puede ser estático o dinámico. En cuanto a *specstart* podemos decir que, a grandes rasgos, se encarga de inicializar los estados de los slots de la ventana al estado FREE, inicializa los punteros no especulativo y más especulativo de la ventana, inicializa las matrices de acceso e inidirección, las variables asociadas a la matriz de indirección, así como el vector Global Exposed Load. Además asigna el valor, en tiempo de ejecución, del límite de iteraciones permitido mediante el único argumento de tipo entero que toma la función: *specstart(int iterations)*.

2.2.8. Utilización del motor y ajuste de variables

En este punto, descritas ya tanto las estructuras de datos que forman el motor de paralelización especulativa, como las distintas operaciones que soporta, podemos empezar a utilizarlo, pero antes de nada cabe mencionar cierto tipo de variables específicas para cada problema, y que se implementarán en uno de los ficheros del motor. Estas variables son las siguientes:

- **threads**: indica el número de threads disponibles para el problema.
- **wsize**: tamaño de ventana, es decir, el número de slots que la forman.
- **blk**: cuando el tamaño de bloque es estático, esta variable implementa el tamaño de los bloques, en concreto, el número de iteraciones que los forman.
- **shared_size**: es el tamaño de la estructura de datos compartida sobre la que se realizarán operaciones de lectura y escritura.
- **cur_upper_size**: número máximo de iteraciones del bucle a paralelizar.
- **max_upper_size**: es una variable similar a *cur_upper_size*.

Una vez implementados los valores de estas constantes, se comienza a modificar el código de la aplicación a paralelizar. Lo primero es incluir el fichero que contiene las variables descritas anteriormente y la librería de OpenMP (*omp_lib.h*). Hecho esto se invocan las dos funciones de inicialización del motor *specinit* y *specstart(int iterations)*, y a una función de la librería OpenMP denominada *omp_set_num_threads(int threads)*, para indicar el número de threads que se van a utilizar. Por último hay que indicar al motor que variables se utilizarán, y de que tipo serán, a saber, privadas o compartidas. Existen un número de variables que siempre serán del mismo tipo para todo programa:

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

A partir de aquí, se puede comenzar a implementar la paralelización del bucle en cuestión, para ello nos basaremos en las directivas que proporciona OpenMP. Para realizar tanto lecturas y escrituras, como operaciones reducibles sobre variables especulativas, se debe proceder a sustituir las instrucciones por las funciones descritas en este capítulo.

Para finalizar, cuando la ejecución del thread acaba, se procede a consolidar los datos mediante la función *threadend*, y se comprueba si quedan bloques por ejecutar.

2.2.9. Implementación actual de las operaciones en el motor

Operaciones de lectura y escritura

En la figura 2.17 se muestra la implementación actual de las operaciones de lectura y escritura especulativas en sintaxis de `C`. De esta figura podemos extraer lo siguiente: sólo la primera carga de datos requiere un manejo especial por el protocolo (línea 1 de la figura 2.17(a)). La búsqueda de versiones predecesoras de los datos en cargas sólo requerirá consultar un elemento de `AM` por thread (líneas 6 a 11 de la figura 2.17(a)). De forma similar, la búsqueda de violaciones de dependencia en la escritura de datos sólo requiere consultar un elemento de `AM` por thread (líneas 11 a 15 de la figura 2.17(b)). El uso de la estructura `GlExpLd` evita la búsqueda de violaciones de dependencia cuando ningún thread ha ejecutado una operación de lectura de datos (línea 10 de la figura 2.17(b)). Además, la búsqueda de violaciones de dependencia puede parar antes si se encuentra un thread sucesor que haya modificado los datos sin una lectura previa (líneas 12 y 13 de la figura 2.17(b)).

Cabe mencionar que los `squashes` sólo pueden ser provocados por escrituras cuando se soporta el *forwarding*. La operación `squash()` simplemente modifica el valor en la ventana del motor a `SQUASHED` y mueve el puntero al thread más especulativo hacia atrás. Tras ello, cuando el thread que sufre un squash finaliza, puede provocar la re-ejecución de los hilos afectados. Además, para incrementar las ventajas de indicar el squash tan pronto como sea posible, los threads comprueban su estado de ventana antes de todas las operaciones especulativas de memoria (código no mostrado) e inmediatamente terminan la ejecución si se encuentran en estado de squash. Así, la operación de squash global puede empezar antes y los threads pueden ser reasignados antes.

Operaciones de commit

La figura 2.18 muestra una implementación del código ejecutado por cada thread al final de su ejecución en sintaxis de C. Este código está dividido principalmente en dos secciones: el commit propiamente dicho (líneas 3 a 16) y la asignación de un nuevo thread (líneas 22 a 29). De esta figura, podemos extraer lo siguiente: sólo el thread no especulativo puede realizar commits (línea 3), y es responsable de realizar el commit de sus datos, y el de todos los threads sucesores que hayan finalizado su ejecución (líneas 5 a 14). Esta operación se limita a comprobar los elementos accedidos por los threads, como se identifican en la estructura IM (líneas 10 y 11). Cuando la ventana se completa, los procesadores están en sondeo y parada sin parar hasta que hay un slot libre (línea 22). Finalmente, antes de empezar la ejecución de un nuevo thread, la estructura AM se limpia eficientemente para su re-utilización utilizando

```
 1.  if (AM[I][tid] == NotAcc) {
 2.    GlExpLd[I]=TRUE;
 3.    AM[I][tid]=ExpLd;
 4.    IM[++tail[tid]][tid]=I;
 5.    #pragma memory fence
 6.    for (j=tid-1; j>=non_spec; j--)
 7.      if (AM[I][j] == Mod || AM[I][j] == ExpLdMod) {
 8.        version[I][tid]=version[I][j];
 9.        forwarded=TRUE;
10.        break;
11.      }
12.    if (!forwarded)
13.      version[I][tid]=ref[I];
14.  }
15.  lvalue=version[I][tid];
```

(a)

```
 1.  version[I][tid]=rvalue;
 2.  #pragma memory fence
 3.  if (AM[I][tid] == NotAcc) {
 4.    AM[I][tid]=Mod;
 5.    IM[++tail[tid]][tid]=I;
 6.  }
 7.  if (AM[I][tid] == ExpLd)
 8.    AM[I][tid]=ExpLdMod;
 9.  #pragma memory fence;
10.  if (GlExpLd[I])
11.    for (j=tid+1; j<=most_spec; j++)
12.      if (AM[I][j] == Mod)
13.        break;
14.      else if (AM[I][j] != NotAcc)
15.        {squash(j); break;}
```

(b)

Figura 2.17: Código en C de (a) operación de lectura y (b) operación de escritura. En esta figura `I` es el índice correspondiente al elemento de la estructura al que se está accediendo, `tid` es el thread sobre el que se está operando, `ref` es la estructura de datos original, y `lvalue` y `rvalue` corresponden a las variables utilizadas en las operaciones originales.

IM (líneas 24 y 25). Tras incrementar el puntero al elemento más especulativo y almacenar un slot en la ventana deslizante, el procesador está preparado para ejecutar otro thread (código no mostrado para simplificar). En la práctica, la operación `do_squash()` simplemente requiere poner el slot en la ventana como `FREE`.

2.3. Un ejemplo de paralelización especulativa

En este punto mostraremos un ejemplo, paso a paso, de como paralelizar especulativamente una aplicación mediante el motor descrito. Para ello contamos con una aplicación muy sencilla que cuenta con una operación de lectura y otra de escritura especulativas. Este ejemplo, desarrollado en lenguaje C, está incluido en [7]. Sin más preámbulos veamos la aplicación en formato secuencial.

La aplicación secuencial

El código C de la aplicación secuencial es el siguiente:

```
 1 // APLICACION SINTETICA escrita en C //
 2 // Requisitos: supone los valores de entrada mayores que 0 //
 3
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6 #include "miejemplovariables.h"
 7
 8 int main()
 9 {
10
11 // Variables locales
```

```
1.  #pragma critical
2.  if (window[tid] != SQUASHED) {
3.    if (tid == non_spec) {
4.      window[tid]=DONE;
5.      for (i=non_spec; i<=most_spec; i++) {
6.        if (window[i] == DONE &&
              window[i+1] != DONE)
7.          {last=i; break;}
8.      }
9.      for (j=non_spec; j<=last; j++) {
10.       for (k=1; k<=tail[j]; k++)
11.         ref[IM[k][j]]=
              version[IM[k][j]][j];
12. #pragma memory fence
13.       window[j]=FREE
14.     }
15.     non_spec=last+1;
16.   }
17.   else
18.     window[tid]=DONE;
19. }
20. else do_squash();
21. #pragma end critical
22. while(window[most_spec+1] != FREE) {}
23. #pragma critical
24. for (j=1; j<=tail[most_spec+1]; j++)
25.   AM[IM[j][most_spec+1]][most_spec+1]=NotAcc;
26. #pragma memory fence
27. tail[most_spec+1]=0;
28. window[most_spec+1]=RUN;
29. most_spec++;
30. #pragma end critical
```

Figura 2.18: Código en C de la operación de commit. En esta figura `I` es el índice correspondiente al elemento de la estructura al que se está accediendo, `tid` es el thread sobre el que se está operando, `ref` es la estructura de datos original, y `lvalue` y `rvalue` corresponden a las variables utilizadas en las operaciones originales.

```
12 // P indica la iteraccion en la que estamos, Q indica el indice del vector
13
14   int P, Q, aux, i;
15   FILE *fichero;
16   int sumatorio=0;
17
18 // Apertura y lectura del fichero
19
20   if ((fichero = fopen("rand1000000.in", "r")) == NULL)
21   {
22     printf ( "Error en apertura del fichero para lectura \n " );
23     exit(0);
24   }
25   else
26   {
27     fscanf(fichero, "%d", &aux);
28     for ( i = 0 ; i < MAX ; i++)
29     {
30       vector[i] = aux;
31       fscanf(fichero, "%d", &aux);
32     }
33     fclose (fichero);
34   }
35
```

```
36 // miejemplocode: debug
37 //  for ( i = 0; i < MAX; i++)
38 //    printf("%d\n", vector[i]);
39 // miejemplocode: end debug
40
41 // El bucle no es paralelizable porque los indices
42 // de los elementos del vector donde escribimos
43 // dependen de los valores de la entrada del fichero
44 // ESTO ES LO QUE SE BUSCA
45 // EMPEZAMOS
46
47   for ( P = 1 ; P <= NITER ; P++ )
48   {
49 //  Codigo del bucle
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 // Escritura del resultado del vector
62   printf(" Resultado del vector\n");
63   for ( i = 0; i < MAX; i++)
64     sumatorio = sumatorio + vector[i];
65   printf("%d\n",sumatorio);
66 }
```

Lo primero que se ha realizado, es un programa que genero números aleatorios. Mediante este programa se genera un fichero con un millón de números. En nuestro código de las líneas 18 a 34 se lee dicho fichero con los números que han sido obtenidos aleatoriamente. Sin embargo, es importante indicar que aunque los números sean aleatorios, la entrada siempre será la misma, es decir, el mismo fichero, para poder repetir el experimento varias veces y de esta forma obtener el mismo resultado y que por tanto sean comparables. Cabe indicar que, para que todo funcione correctamente, los números del fichero deben ser positivos.

Una vez leído el fichero y guardados sus datos en la variable `vector`, comienza el bucle principal con `NITER` iteraciones, desde la línea 47 a la 59. Aquí se lee un elemento del vector (línea 53) y se escribe en otro elemento del vector (línea 57). La variable `Q` almacena el índice del elemento del vector a leer y escribir:

- Para la lectura, el valor de `Q` es el resto entre la iteración actual en que la se encuentra y el tamaño máximo del vector, más uno.
- Para la escritura, el valor de `Q` es el resto de cuatro veces el valor leído del vector y el tamaño máximo, más uno.

Podemos comprobar que el valor de la variable `Q` se desconoce en tiempo de compilación, porque depende del valor leído del vector, que a su vez depende del valor correspondiente en el fichero de entrada. Es por estos motivos que el compilador no puede garantizar la ejecución concurrente de varias iteraciones sin errores, por tanto no paraleliza el bucle. Es aquí dónde podemos encuadrar el motor descrito, porque ofrece una solución a algunos de los problemas no paralelizables por un compilador convencional, y este algoritmo es un ejemplo perfecto.

Por último el código realiza una operación de comprobación de las líneas 61 a 65, dónde emite la suma de los elementos de `vector`, para comprobar si los datos son correctos.

Antes de empezar a explicar el proceso de paralelización especulativa de este código, es conveniente describir el significado de las variables empleadas en la aplicación sintética:

- *NITER*: es el número de iteraciones que se repite el bucle.
- *MAX*: es el tamaño del vector.
- *aux*: se utiliza para guardar el valor de un elemento del vector.
- *Q*: variable que guarda el índice del vector.
- *P*: variable que indica la iteración actual del bucle.
- *sumatorio*: es la suma de los elementos de vector al final de la ejecución, sirve para realizar comprobaciones, no interviene en el código a paralelizar.

Paralelización especulativa de la aplicación secuencial

A la hora de paralelizar un código secuencial de forma especulativa existen una serie de preguntas que deben ser respondidas:

¿Qué líneas de código paralelizar especulativamente? Lo primero que es necesario conocer es qué líneas de código pueden ser paralelizadas. En este ejemplo concreto es muy sencillo obtenerlas, debido a que solamente hay un bucle. En caso de que hubiera varios bucles, se podrán paralelizar siempre y cuando no estén anidados. Además será recomendable que el bucle tenga un tamaño relativamente grande para poder obtener un buen rendimiento. En el ejemplo no se persigue obtener buenos rendimientos, sino simplemente mostrar el modo de paralelizar una aplicación.

En el caso de bucles anidados sólo puede paralelizarse uno de los bucles del anidamiento: el más externo, el más interno, o algún bucle intermedio.

¿Cuales son las variables especulativas? Es decir, cuáles son las variables compartidas dentro del bucle que pueden dar lugar a violaciones de dependencia durante la ejecución paralela del bucle. En este caso dentro del bucle se emplean cuatro variables: `P`,`Q`,`aux` y `vector`. Las tres primeras modifican su valor al principio del bucle antes de ser utilizadas, por lo que no provocan dependencias y se consideran variables privadas, mientras que la cuarta variable es compartida por todas las iteraciones del bucle y podría ocurrir que mientras la iteración `i`, estuviera escribiendo en la posición `k` del vector, la iteración `i-1` estuviera leyendo en esa misma posición `k`, dando lugar a una violación de dependencia. Por lo tanto, el candidato a *variable especulativa* es la variable `vector`.

¿Cual es el tamaño de la variable especulativa? Una vez conocida cuál es la variable especulativa, se necesita conocer su tamaño. Para ello se ha implementado un fichero denominado `miejemplovariables.h`.

```
1 // En este fichero se incluyen todas la variables comunes de miejemplo
2
3 //Constantes
4 //Tamaño del vector y numero de iteraciones
5       #define MAX 100
6       #define NITER 1000000
7
8 //Estructuras de datos
9       int vector[MAX];
```

El tamaño de `vector` es `MAX`, concretamente 100.

Respondidas estas tres cuestiones se puede comenzar la modificación de la aplicación secuencial. El primer paso consiste en añadir nuevos ficheros en la ruta de la aplicación:

- `speccode-vXX.c`. Contiene el código del motor especulativo en su versión `XX`.
- `variables.h`. Este fichero tiene unos parámetros de configuración que dependen de la aplicación concreta a paralelizar y que se emplean para indicar al motor especulativo el tamaño del bucle y el tamaño de las variables especulativas. Por tanto, es necesario modificar el fichero `variables.h` como se muestra a continuación:

```
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
```

La variable `threads` indica el número de threads disponibles. El número de slots que forman la ventana deslizante se indica mediante `wsize`. El número de iteraciones que forman cada bloque que ejecutará un thread, cuando el tamaño de los bloques sea estático, es indicado por `blk`. La variable `shared_size` indica al motor el tamaño de las variables especulativas. En este caso, sólo se tiene la variable `vector`, cuyo tamaño es 100. La variable `cur_upper_limit` indica el número de iteraciones que hay que ejecutar. En este caso coincide con el valor de `NITER`. La variables `max_upper_limit` como se ha comentado a lo largo de este capítulo, es similar a `cur_upper_limit`.

Una vez configurados los parámetros del motor, se modifica el código de partida, añadiendo en primer lugar las dos cabeceras siguientes en el fichero `ejemplo-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"
```

A continuación se añade una línea para inicializar las estructuras del motor:

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

Justo antes del comienzo del bucle se insertan las siguientes líneas que están relacionados con el funcionamiento de OpenMP (salvo la función del motor especulativo `specstart(NITER)`):

```
60 //    speccode: OMP threading directive
61       omp_set_num_threads(threads);
62 //    speccode: initializing speculation structures
63       specstart(NITER);
...
70 // Inicio de la parte paralela
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)
```

La línea 61 indica el número de threads que van a estar trabajando en paralelo y la línea 63 inicializa las estructuras especulativas del motor, indicándolas el número de iteraciones que se van a producir.

La línea 74 es una directiva OpenMP, que marca el inicio del bucle `for` que va a paralelizarse. De la línea 74 a 79 se encuentran las cláusulas de la directiva `parallel for`. Lo que encontramos en la línea 74 `default(none)` indica que la clasificación de las variables en privadas y compartidas se hará mediante las cláusulas correspondientes. La mayoría de las variables que aparecen son variables propias del motor, exceptuando `P, Q, aux` y `vector` de las que ya se ha hablado anteriormente: las tres primeras son variables privadas, mientras que la última es compartida. Por último, la línea 83 indica que el tamaño de los bloques de iteraciones será constante y no variará a lo largo de la ejecución.

A continuación comienza el bucle y se sustituye la línea de código secuencial:

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

por

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

De este modo se ejecutará un bucle en paralelo con `P` como índice, con iteración inicial 1 y con identificador del bucle 1.

El siguiente paso es buscar en el bucle secuencial todas las líneas en las que intervenga la variable especulativa `vector`. Esta variable aparece en las líneas 53 y 57 del código secuencial. La primera aparición es una lectura y por tanto se sustituye por la función **specload** del motor especulativo:

```
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 //************************************************************************
```

El significado de cada uno de los argumentos de la función puede verse en la sección anterior. En lo que concierne a las variables de la aplicación concreta, han de pasarse a la función la estructura especulativa `vector` y el índice `Q` que indica que entrada de la estructura quiere leerse, por comodidad, se pasa como valor la variable linear, a la que previamente le asignamos el valor correspondiente, `Q-1`. Además, en la función *specload* se realiza una comprobación para ver si se ha detectado una situación de *early squashing*. Si se ha producido se reinicia la ejecución de ese bloque de iteraciones.

La segunda de las apariciones de la variable `vector` es una escritura sobre ella y debe sustituirse por la función **specstore** del motor:

```
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);
//************************************************************************
```

El significado de cada uno de los argumentos de la función puede verse en la sección anterior. En lo que concierne a las variables de la aplicación concreta, se deben pasar el índice `Q` que indica que entrada de la estructura quiere escribirse, por comodidad, se pasa como valor la variable linear, a la que previamente le asignamos el valor correspondiente, `Q-1`. También se le pasa el valor a almacenar, en este caso `aux`, que por las mismas razones que en el caso anterior se sustituye por la variable `value`.

Una vez sustituidas todas las lecturas y escrituras por sus correspondientes funciones del motor se debe añadir al final del bucle la sentencia:

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

Una vez realizado todo esto, la aplicación ha quedado preparada para ser paralelizada por el motor especulativo, por lo que sólo necesita ser compilada y ejecutada por el número de threads que corresponda.

Resumen

Todos los pasos realizados para paralelizar especulativamente una aplicación se pueden resumir en:

1. Identificar el bucle que se quiere paralelizar y su número de iteraciones.
2. Identificar las variables especulativas del bucle y su tamaño total.
3. Configurar el fichero `variables.h` con el número de iteraciones y el tamaño de las variables especulativas.
4. Añadir las cabeceras de OpenMP y del motor especulativo.
5. Inicializar las estructuras del motor y declarar las variables compartidas y privadas del bucle mediante las directivas OpenMP.
6. Introducir una sentencia de inicio de bucle especulativo.
7. Sustituir las lecturas y escrituras de las variables especulativas por `specload` y `specstores` respectivamente.
8. Introducir una sentencia de fin de bucle especulativo.

Capítulo 3

Limitaciones del motor especulativo original

La versión original del motor especulativo cuenta con ciertas restricciones. A continuación, abordaremos cuáles son, y enumeraremos una serie de ideas para obtener una versión del motor nueva que subsane estas limitaciones.

La versión original del motor cuenta con una serie de condiciones que limitan su utilización. A continuación citaremos alguna de ellas:

- Sólo podemos trabajar con **una sola estructura**: la versión original del motor solamente permite la ejecución especulativa de los datos de una estructura, imposibilitando paralelizar especulativamente aplicaciones que utilicen más de una estructura de datos.
- El motor **no especula sobre variables sueltas**: si queremos ejecutar especulativamente un programa sin tener que agrupar las variables a paralelizar en una estructura de datos (porque pueden no tener nada en común) la versión actual no nos lo permite.
- El motor original **no especula con estructuras de datos distintas de vectores y matrices**.
- La versión original del motor sólo permite especular con **datos de un mismo tipo**: char, int, double, etc. Sin embargo, no es capaz de ejecutar especulativamente bucles que contengan variables especulativas de distinto tipo.

El estudio de estas limitaciones y de otras propuestas realizadas por grupos de investigación llevó a los autores de este Trabajo a desarrollar un sistema de ejecución especulativa que se basa en el paso por referencia de las variables a modificar especulativamente, en lugar de "por copia" como se hacía en la solución original.

Tras sopesar cómo se podría realizar una nueva implementación que soporte las características deseadas,se han encontrado tres posibles soluciones:

- Solución 1: basada en matrices de versión de datos y matrices de punteros para cada thread.

- Solución 2: basada en una matriz global de punteros y matrices de versión de datos para cada thread.
- Solución 3: basada en matrices dinámicas de punteros.

3.1. Solución 1: basada en matrices de versión de datos y matrices de punteros para cada thread

Se podría implementar una estructura auxiliar con N filas (tantas como elementos), y dos columnas, una que indique la posición de memoria del puntero, y otra con su valor para evitar sucesivos accesos a memoria.

+ Se conseguiría un acceso rápido en ejecución a los valores de los punteros.
− Necesarias nuevas estructuras, luego más espacio en memoria.
− La búsqueda de datos supondría una búsqueda elemento por elemento, en lugar de la búsqueda directa de la forma antigua.

3.2. Solución 2: basada en una matriz global de punteros y matrices de versión de datos para cada thread

Podría pensarse en implementar un único vector, que contenga todas las estructuras referenciadas, una a continuación de la otra.

+ Se conseguiría un modo de soportar varias estructuras de datos de manera sencilla.
− Numeración: ¿Cómo numeramos los elementos de la nueva estructura?.

3.3. Solución 3: basada en matrices dinámicas de punteros

Podríamos implementar una matriz de punteros para cada elemento de la ventana deslizante cuyo tamaño fuese incrementando en función de las necesidades de la aplicación en cuestión.

+ Se conseguiría que la memoria se reservase sólo cuando fuese necesaria.
+ Podríamos cambiar el tamaño de la matriz de una posición de la ventana sin que las demás fuesen afectadas.
− La búsqueda de datos supondría una búsqueda elemento por elemento, y matriz por matriz de posición en posición.
− La reserva de memoria para incrementar el tamaño de las matrices aumentará el tiempo de ejecución.

La decisión de cuál es la solución más apropiada requiere un estudio más detallado de las ventajas e inconvenientes de cada una de ellas. Dicho estudio se realizará en los capítulos 6, 7 y 8.

Capítulo 4

Gestión de Proyecto

En este capítulo están descritos los pasos necesarios para llevar a cabo este Proyecto. Para ello se dará un modelo de desarrollo software, una planificación del proyecto, con su respectivo seguimiento, y el presupuesto necesario para el desarrollo.

4.1. Modelo de desarrollo

Sommerville [16] define modelo de proceso de software como: *una representación simplificada de un proceso de software, representada desde una perspectiva específica. Por su naturaleza los modelos son simplificados, por lo tanto un modelo de procesos del software es una abstracción de un proceso real.*

Los modelos genéricos no son descripciones definitivas de procesos de software; sin embargo, son abstracciones útiles que pueden ser utilizadas para explicar diferentes enfoques del desarrollo de software. De este modo en este proyecto seguiremos un modelo de desarrollo teniendo en cuenta el carácter especial del mismo, dado que no es un proyecto al uso, sino que se trata de un trabajo de investigación, esto hace que no sea eficiente utilizar el desarrollo de Proceso Racional Unificado (RUP), y descarta los casos de uso para cubrir los objetivos del Trabajo de Fin de Grado. A continuación describiremos brevemente los modelos que se sopesaron llevar a cabo y los motivos de ello.

- El primer modelo que se nos ocurrió, fue uno de los más simples, **codificar y corregir (code-and-fix)**, este método permitía avanzar el código del proyecto de forma rápida, obteniendo unas primeras versiones del motor deseado, en las fases iniciales del desarrollo. Este método se ajusta de forma bastante exhaustiva a nuestro proyecto dado que no tiene una envergadura demasiado extensa (menos de 10 000 líneas de código). Sin embargo, se desechó dado que un número muy elevado de correcciones, pueden hacer que el código sea ilegible. Además, en un proyecto de este tipo codificar sin hacer un análisis del problema y sin estudiar la solución, puede conducir a soluciones muy poco robustas y seguramente lentas.

- Otro modelo que se sopesó fue el **desarrollo evolutivo**, de modo que se implementasen versiones preliminares del código, para mostrárselas al cliente, en este caso los tutores, y así refinar el producto hasta obtener un software correcto. En concreto la vertiente

de desarrollo evolutivo seleccionada fue el `desarrollo exploratorio`, de modo que el sistema evolucionaría en función de las características propuestas. Sin embargo, este método tenía el mismo inconveniente que el anterior, y es que demasiados cambios pueden hacer del código algo indescifrable. Pero cabe indicar que este método es adecuado para proyectos de poco tamaño, como es el caso.

Vistas las opciones escogidas, la decisión fue decantarse por el **desarrollo evolutivo**. A continuación describiremos los motivos: este proyecto contiene relativamente pocas líneas de código, sin embargo, esto no tiene por qué provocar que la estructura final del código tenga sucesivas correcciones, con lo que el producto final podría resultar ilegible. Para intentar evitar el principal inconveniente que acompaña a este modelo se intentarán seguir una serie de pautas (comentarios en el código, control de versiones, etc.) para evitar obtener un código ininteligible.

Otra de las razones es que se considera que los requisitos no son demasiado estrictos en este trabajo. Prueba de ello es que el objetivo no es la obtención de un producto final inicialmente conocido, sino que al tener una naturaleza de investigación, los resultados finales son desconocidos totalmente, y puede que no se ajusten a las previsiones realizadas inicialmente. Por tanto sería deseable que los objetivos del proyecto pudiesen cambiar fácilmente a lo largo del desarrollo del mismo, cosa que se adapta perfectamente a la filosofía de este modelo.

Además con el desarrollo evolutivo podemos generar prototipos del producto final para ayudarnos en la obtención de resultados que marcarán la pauta a seguir para obtener conclusiones del producto. En cuanto a los riesgos, se puede decir que este modelo no presta demasiada atención a su gestión, sin embargo, no es necesario por los motivos ya mencionados, y es que no se conoce el resultado obtenido, por tanto las conclusiones pueden ser totalmente contrarias a lo esperado, pero no por ello serán unas conclusiones inválidas.

4.2. Planificación temporal

Esta sección especifica la planificación del proyecto, en concreto, trata de proporcionar unas pautas a seguir en el desarrollo del mismo. Esta guía se realizará acorde al modelo de desarrollo escogido, por tanto, está muy expuesta a modificaciones a lo largo del proceso de desarrollo, sin embargo, es conveniente fijar una planificación inicial.

A continuación se expondrá la planificación realizada al comienzo de este proyecto, por ello, se corresponde con una planificación de alto nivel, en la que se especifican la duración de las diferentes versiones (evoluciones) del proyecto. En el cuadro 4.1 se muestra la planificación inicial, mientras que en la figura 4.1 podemos ver el diagrama de Gantt correspondiente a dicha planificación.

A continuación se realizará un pequeño resumen de los objetivos que se han de cumplir en cada una de las evoluciones del proyecto:

- **Introducción a los conceptos teóricos**. Esta fase se caracteriza por la toma de contacto con la paralelización especulativa y con el motor de paralelización especulativa `SpecEngine`. Además se detallarán los principales requisitos y los objetivos del proyecto.

  Para comprobar que se han cumplido los objetivos de esta etapa se utilizarán los siguientes criterios de evaluación:

  - ¿Se tiene el suficiente conocimiento teórico sobre la paralelización especulativa como para comenzar la siguiente etapa?

Id	Nombre de tarea	Esfuerzo (Horas Persona)	Comienzo	Fin	Predecesoras
1	<Inicio>	0	lun 03/10/2011	lun 03/10/2011	
2	**Introducción a los conceptos teóricos**	**110**	**lun 03/10/2011**	**mié 02/11/2011**	**1**
3	Estudio de las características de la paralelización especulativa	50	lun 03/10/2011	vie 14/10/2011	
4	Estudio de las características del motor de paralelización especulativa "`SpecEngine`"	25	lun 17/10/2011	vie 21/10/2011	3
5	Descripción de los objetivos del proyecto	5	lun 24/10/2011	vie 24/10/2011	4
6	Descripción del alcance del del proyecto	5	lun 25/10/2011	vie 25/10/2011	5
7	Elicitación de los requisitos principales del proyecto	25	lun 26/10/2011	mié 02/11/2011	6
8	**Obtención de soluciones teóricas**	**450**	**jue 03/11/2011**	**mié 07/03/2012**	**2**
9	Obtención de una solución posible	90	jue 03/11/2011	mié 30/11/2011	
10	Análisis de la solución	40	jue 01/12/2011	mié 14/12/2011	9
11	Análisis de costes de sus operaciones	15	jue 15/12/2011	lun 19/12/2011	10
12	Elección de una solución (tras varias iteraciones de las tareas 9, 10 y 11)	15	mar 20/12/2011	jue 22/12/2011	11
13	**Elaboración de una versión del motor especulativo que soporte aritmética de punteros**	**150**	**jue 08/03/2012**	**mié 18/04/2012**	**8**
14	**<Obtención de una implementación del nuevo motor especulativo>**	**0**	**mié 18/04/2012**	**mié 18/04/2012**	**13**
15	**Pruebas**	**25**	**jue 19/04/2012**	**jue 26/04/2012**	**14**
16	Distinto número threads	10	jue 19/04/2012	mié 25/04/2012	
17	Distinto tamaño de bloque	15	jue 19/04/2012	jue 26/04/2012	
18	**Experimentación**	**50**	**vie 27/04/2012**	**lun 14/05/2012**	**15**
19	Ejemplo sintético	15	vie 27/04/2012	lun 14/05/2012	
20	Menor círculo contenedor	20	vie 27/04/2012	lun 14/05/2012	
21	Envolvente convexo	15	vie 27/04/2012	lun 14/05/2012	
22	**<Fin>**	**0**	**lun 14/05/2012**	**lun 14/05/2012**	**18**
	TOTAL	**785**	**lun 03/10/2011**	**lun 14/05/2012**	

Cuadro 4.1: Planificación inicial del proyecto.

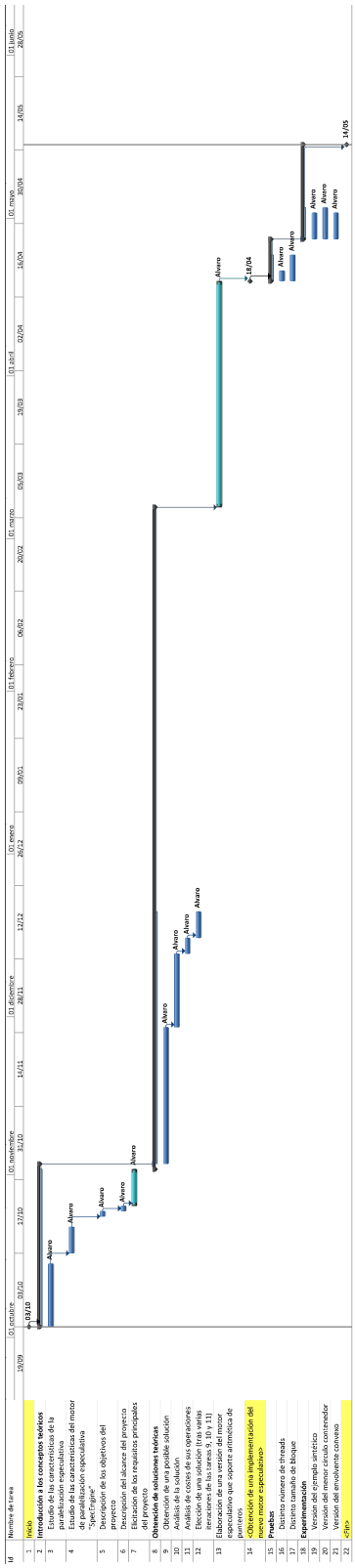

Figura 4.1: Diagrama de Gantt de la planificación inicial.

- ¿Se tiene una noción suficiente de la arquitectura del motor especulativo?
- ¿Se han definido claramente el alcance y los objetivos del proyecto?
- ¿Se han obtenido los principales requisitos del proyecto?

- **Obtención de varias soluciones teóricas**. Esta etapa será una de las más largas del proyecto, en concreto comprenderá el análisis de varias soluciones factibles que resolviesen el problema principal del proyecto de diferentes formas. La etapa estará formada por varias iteraciones, en las que tendrán lugar desde el surgimiento de una idea que solucione el problema, hasta la descripción general de la misma y el análisis de las operaciones. Una vez tengamos varias soluciones factibles, haremos una comparación y seleccionaremos la que nos parezca mejor.

  Para realizar la estimación del tiempo de esta fase nos pondremos en la tesitura de una planificación pesimista, es decir, contaremos con que esta fase nos llevará más tiempo del que pueda parecer para evitar retrasos en la entrega. Hemos de tener en cuenta además la época de exámenes donde se descuidará en gran medida el trabajo en el proyecto.

  Al igual que en la etapa anterior, utilizaremos ciertos criterios de evaluación con el objetivo de asegurarnos de que se cumplen los objetivos de esta etapa. Los criterios que utilizaremos son:

  - ¿Se tiene un análisis suficientemente descriptivo de las versiones desarrolladas?
  - ¿Existe una comparación concisa de las soluciones que nos permita elegir una de ellas?
  - ¿Se han obtenido todos requisitos del proyecto?

- **Elaboración de una versión del motor que soporte punteros**. Durante esta etapa realizaremos la implementación de la solución elegida. Esta etapa no debería de llevarnos demasiado tiempo, puesto que nuestra planificación contempla un análisis exhaustivo de la solución, y por tanto, el diseño de la misma debería estar muy detallado, y no retrasarnos mucho tiempo.

  Para comprobar el correcto desarrollo de esta etapa también tenemos los siguientes criterios de evaluación:

  - ¿Se tiene una implementación del motor lo suficientemente robusta?
  - ¿Se ha documentado correctamente cómo se ha desarrollado la implementación?

- **Pruebas**. Se realizarán las pruebas pertinentes con la solución del motor implementada. Para ello se elaborará la versión especulativa de una aplicación que someta a una gran carga al motor, es decir, que la someta a múltiples operaciones, en multitud de escenarios. Con esto podremos comprobar si la nueva versión es correcta, robusta, etc., es decir, la validez del código.

  En esta la etapa recién descrita los criterios de evaluación son:

  - ¿El resultado de las pruebas ha sido satisfactorio?
  - ¿Hemos elaborado una versión del motor especulativo que funciona en la mayoría de las situaciones?

- ¿La batería de pruebas ha sido suficientemente amplia?

- **Experimentación**. En esta etapa mediremos experimentalmente los tiempos de ejecución del nuevo motor para comprobar sus características respecto al original. Para ello se elaborarán versiones especulativas de diferentes aplicaciones de las que disponemos de una versión especulativa con el motor original. Así podremos cerciorarnos de si la nueva arquitectura del motor es más rápida que las anteriores, si utiliza menos memoria, etc., es decir, comprobar cuestiones de eficiencia.

  En esta última etapa del proyecto los criterios de evaluación son:

  - ¿Hemos elaborado una versión del motor especulativo que funciona en la mayoría de las situaciones?
  - ¿Se han obtenido conclusiones de algún tipo sobre el nuevo motor especulativo?
  - ¿Se ha cumplido los objetivos del proyecto?
  - ¿Se han cumplido todos los requisitos del proyecto?

4.2.1. Análisis de riesgos

A continuación se detallarán los riesgos que se identifican en la fase inicial del proyecto. Todos los riesgos identificados serán analizados en los siguientes aspectos:

- **Descripción.** Breve descripción textual.
- **Magnitud.** Estimación de la importancia de sus efectos en caso de que se convierta en un hecho. Se evalúa como muy baja, baja, media, alta, muy alta o catastrófica.
- **Impacto.** Descripción textual de los efectos sobre el proyecto de la transformación del riesgo en un hecho.
- **Indicadores.** Magnitudes a observar para intuir la aparición del riesgo.
- **Plan de acción.** Medidas a tomar en el proyecto para evitar la aparición del riesgo o minimizar su futuro impacto, aplicadas antes de que el riesgo se convierta en un hecho.
- **Plan de contingencia.** Medidas a tomar en el proyecto una vez que el riesgo se haya transformado en un hecho.
- **<RSK-01>No cumplir con la fecha de entrega**
  - **Descripción:** el desarrollo del proyecto puede retrasarse y no cumplir con las fechas previstas de finalización.
  - **Magnitud:** muy alta.
  - **Impacto:** retraso en la presentación del proyecto, y por tanto en la obtención del título. Además se imposibilitaría presentar el proyecto a congresos con una fecha más o menos similar a la planificada.
  - **Probabilidad:** 40 %.
  - **Indicadores:** cuando se acerque la fecha planificada de entrega del proyecto y no tengamos una versión estable.

- **Plan de acción:** se intentará seguir una pauta de trabajo en la que se estudien a fondo los problemas y ventajas de una solución, para evitar implementar soluciones no deseadas, descartándolas en el análisis.
- **Plan de contingencia:** retrasar la fecha de lectura del proyecto.

- **<RSK-02>Carencia de tiempo para realizar el proyecto**[1]
  - **Descripción:** como no se realiza únicamente el proyecto, puede ser que las demás tareas impidan el desarrollo constante de la solución.
  - **Magnitud:** media.
  - **Impacto:** retraso en el proyecto.
  - **Probabilidad:** 50 %.
  - **Indicadores:** cuando la carga de trabajo sea muy alta y se descuide el proyecto.
  - **Plan de acción:** evitar que se acumulen los trabajos, tanto ajenos al proyecto, como internos.
  - **Plan de contingencia:** si una semana es muy difícil sacar tiempo para avanzar el proyecto, intentar dedicar un poco más en las siguientes para compensarlo.

- **<RSK-03>No cumplir el objetivo final del proyecto**
  - **Descripción:** el proyecto pretende servir para obtener un motor de paralelización especulativa que soporte la aritmética de punteros, sin embargo, no estamos seguros al 100 % de que sea posible.
  - **Magnitud:** muy alta.
  - **Impacto:** cambio en los objetivos del proyecto, hasta convertirlo en un proyecto de los análisis realizados en la búsqueda de la solución.
  - **Probabilidad:** 15 %.
  - **Indicadores:** si se repiten ciclos de llegar a una solución, y que sea imposible implementarla de forma correcta.
  - **Plan de acción:** se intentarán obtener soluciones coherentes e implementables.
  - **Plan de contingencia:** describir en el proyecto los problemas acaecidos en las distintas soluciones elaboradas.

- **<RSK-04>Mala planificación**
  - **Descripción:** la planificación puede ser errónea.
  - **Magnitud:** media.
  - **Impacto:** retraso en el proyecto.
  - **Probabilidad:** 70 %.
  - **Indicadores:** si realizando el seguimiento del proyecto comprobamos que las fechas sufren variaciones importantes.

[1] Este riego guarda una relación directamente proporcional con el riesgo <RSK-01>, sin embargo, vemos oportuno incluir ambos dado que hay una etapa que requiere que surjan ideas, y esto puede no ser una cuestión temporal.

  - **Plan de acción:** realizar un seguimiento semanal de la planificación para ajustar la duración de las distintas tareas.
  - **Plan de contingencia:** reajustar la planificación y el tiempo asignado a las distintas tareas para conseguir el objetivo final en la fecha indicada. Llegando a aumentar el número de horas dedicadas al proyecto si fuese necesario.

- **<RSK-05>Plan de pruebas incompleto**
  - **Descripción:** la batería de pruebas realizada en el proyecto no ha contemplado casos críticos o situaciones de posibles fallos.
  - **Magnitud:** media.
  - **Impacto:** puede que el motor no funcione correctamente en todos los casos.
  - **Probabilidad:** 20 %.
  - **Indicadores:** los resultados obtenidos con ciertas aplicaciones no son correctos.
  - **Plan de acción:** utilizar métodos y técnicas formales para la realización de las pruebas (prueba del camino básico, partición de equivalencia, análisis de valores límite, etc.).
  - **Plan de contingencia:** rehacer el plan de pruebas y realizar las pruebas ajustándose al nuevo plan.

- **<RSK-06>Pérdida de información del proyecto, debido a una mala gestión de las copias de seguridad.**
  - **Descripción:** pérdida de los artefactos obtenidos en las etapas previas del proyecto o pérdida de alguna de las versiones.
  - **Magnitud:** muy alta.
  - **Impacto:** pérdida de datos importantes del proyecto y retraso del trabajo.
  - **Probabilidad:** 10 %.
  - **Indicadores:** no presencia de alguna de las versiones anteriores del proyecto. Mala gestión de las copias de seguridad de los documentos.
  - **Plan de acción:** Realizar varias copias de seguridad de todos los elementos implicados en el proyecto, ubicados en diferentes lugares de almacenamiento.
  - **Plan de contingencia:** volver a realizar las partes del proyecto implicadas en la pérdida de información.

- **<RSK-07>Desconocimiento del lenguaje de programación.**
  - **Descripción:** podría darse el caso de no conocer algunos de los detalles del lenguaje de programación seleccionado para la implementación del proyecto.
  - **Magnitud:** muy baja.
  - **Impacto:** retraso en la planificación temporal estimada.
  - **Probabilidad:** 10 %.
  - **Indicadores:** desconocimiento de la implementación de alguno de los conceptos teóricos en el lenguaje `C`.

  - **Plan de acción:** buscar bibliografía que contenga información sobre los detalles del lenguaje `C`, y estudiar la parte correspondiente a las dudas acontecidas.
  - **Plan de contingencia:** retrasar la implementación de la solución hasta que se dominen los conocimientos necesarios para obtenerla.

- **<RSK-08>Elaboración de un código fuente ilegible.**
  - **Descripción:** en fuentes bibliográficas, hemos podido saber que el modelo de desarrollo elegido es propenso a crear desarrollos con muchas correcciones. Esto puede hacer que el código final sea ininteligible.
  - **Magnitud:** alta.
  - **Impacto:** poca reutilización del nuevo motor especulativo.
  - **Probabilidad:** 20 %.
  - **Indicadores:** si ojeamos el código elaborado y nos cuesta mucho entender el significado del mismo.
  - **Plan de acción:** intentar insertar comentarios en todos los lugares del código donde la comprensión del mismo sea farragosa.
  - **Plan de contingencia:** estudiar a fondo los detalles de la implementación que no se comprenda, e intentar elaborar nuevo código fuente reutilizable.

4.3. Seguimiento de la planificación

En este apartado se realizarán las observaciones de la planificación inicial, es decir, si esta ha resultado más o menos correcta, si se han producido retrasos y los motivos. Para ello se verá lo ocurrido en los diferentes objetivos de la planificación inicial. A lo largo del seguimiento temporal podremos encontrar el **seguimiento de los riesgos**.

- **Introducción a los conceptos teóricos**. La fase inicial era para asimilar los conceptos teóricos relativos al proyecto, muchos de ellos ya eran conocidos de un proyecto previo y tan sólo hubo que recordarlos. Por ello, no se precisó de tiempo adicional al planificado, y se cumplieron los objetivos al completo en el tiempo que se predijo.

  Los criterios de evaluación descritos para esta etapa son los siguientes (a continuación de cada criterio se exponen los resultados obtenidos al valorar cada uno de los puntos):

  - ¿Se tiene el suficiente conocimiento teórico sobre la paralelización especulativa como para comenzar la siguiente etapa? *Sí, además de conocer de un proyecto anterior los conceptos generales de este tipo de paralelización, se han realizado varias lecturas complementarias para consolidar los conocimientos.*
  - ¿Se tiene una noción suficiente de la arquitectura del motor especulativo? *Sí, puesto que se ha realizado un proyecto previo donde se utilizó el motor citado.*
  - ¿Se han definido claramente el alcance y los objetivos del proyecto? *Sí.*
  - ¿Se han obtenido los principales requisitos del proyecto? *Sí, se ha elaborado una lista con los principales requisitos que debemos tener en cuenta.*

- **Obtención de varias soluciones teóricas**. Esta fase abordaba el problema principal del proyecto, y por tanto, se predijo que iba a ser larga, haciendo una previsión muy pesimista. Esto fue una decisión acertada ya que como se puede comprobar en el proyecto, se elaboraron varias soluciones teóricas, es decir, varios prototipos del motor, teniendo que realizar varias evoluciones de los mismos. Sin embargo aún teniendo una planificación conservadora, no ha sido correcta, teniendo lugar el riesgo <RSK-04>. Esto ha provocado un retraso de unas noventa horas adicionales sobre lo planificado. A continuación describiremos el seguimiento de cada solución.

  - **Análisis de la solución basada en matrices de versión de datos y matrices de punteros para cada thread**. Esta solución fue la primera que obtuvimos. Por tanto, su obtención fue la que más tiempo nos llevó puesto que era una primera aproximación a la solución deseada. De las noventa horas predichas para la obtención de la idea, nos retrasamos en cincuenta horas más. Además las cuarenta horas destinadas a realizar un análisis no fueron suficientes, necesitando diez horas adicionales. El análisis de costes, se llevó a cabo con las horas supuestas inicialmente.
  - **Análisis de la solución basada en una matriz global de punteros y matrices de versión de datos para cada thread**. Esta solución nos llevó menos tiempo que la anterior puesto que tras tener las bases teóricas de la primera solución, ya sabíamos cómo abordar otro tipo de soluciones. Aún así necesitamos más horas de las predichas para obtener la idea, en concreto, diez más. El resto de tareas de esta iteración fueron resueltas en el tiempo planificado inicialmente.
  - **Análisis de la solución basada en matrices de punteros dinámicas**. Esta solución fue mucho más meditada que las anteriores, y al igual que en la primera, se produjo un retraso considerable sobre la planificación inicial. El motivo principal es que era una solución mucho más ambiciosa que las anteriores porque utilizaba memoria dinámica. Esto hizo que fuésemos más precavidos y que nuestro análisis fuese más exhaustivo. En concreto, el tiempo dedicado a la obtención de la solución no sufrió retraso alguno, y se cumplieron los plazos dedicados a su desarrollo. Sin embargo, en el tiempo dedicado al análisis de la solución, necesitamos más tiempo del que inicialmente supusimos, exactamente unas veinte horas complementarias. El análisis de los costes no tuvo retraso alguno.

  Por último cabe comentar que la elección de una solución entre las tres existentes no sufrió retrasos sobre la planificación inicial.

  Los criterios de evaluación descritos para esta etapa son los siguientes (a continuación de cada criterio se exponen los resultados obtenidos al valorar cada uno de los puntos):

  - ¿Se tiene un análisis suficientemente descriptivo de las versiones desarrolladas? *Sí, se ha realizado una descripción en líneas generales de cada solución. Además se ha descrito cómo se llevarían a cabo las principales operaciones que debe realizar un motor de paralelización especulativa.*
  - ¿Existe una comparación concisa de las soluciones que nos permita elegir una de ellas? *Sí, se ha realizado un análisis del coste de las operaciones del motor especulativo para cada solución, y se han dedicado varios días a la obtención de las ventajas y desventajas de implementar cada solución.*

  - ¿Se han obtenido todos requisitos del proyecto? *Sí, se ha completado la lista de requisitos con los nuevos requisitos detectados.*

- **Elaboración de una versión del motor que soporte punteros**. A priori la implementación del código de la solución elegida iba a ser una etapa corta, sin embargo, por el tipo del problema en cuestión, hubo que profundizar más en los conceptos teóricos de la aritmética de punteros del lenguaje elegido, cosa que retrasó la implementación, es decir, tuvo lugar el riesgo <RSK-07> causando un retraso de veinte horas sobre la planificación inicial.

  Los criterios de evaluación descritos para esta etapa son los siguientes (a continuación de cada criterio se exponen los resultados obtenidos al valorar cada uno de los puntos):

  - ¿Se tiene una implementación del motor lo suficientemente robusta? *Sí, se ha comprobado que el motor realiza las operaciones necesarias para paralelizar especulativamente aplicaciones con aritmética de punteros.*
  - ¿Se ha documentado correctamente cómo se ha desarrollado la implementación? *Sí, se han detallado los pasos seguidos a la hora de elaborar el código del nuevo motor.*

- **Pruebas**. Esta etapa fue una de las más importantes, puesto que se llevaron a cabo todas las pruebas para la obtención de un motor resistente a fallos y, en general, correcto. Por ello, y aún con el relativamente elevado tiempo de la planificación inicial (veinticinco horas) se produjeron retrasos de otras veinticinco horas más. Esto fue debido a la aparición de algunos errores algo enmascarados que provocaron un estudio en profundidad del código existente.

  Los criterios de evaluación descritos para esta etapa son los siguientes (a continuación de cada criterio se exponen los resultados obtenidos al valorar cada uno de los puntos):

  - ¿El resultado de las pruebas ha sido satisfactorio? *Sí, porque hemos obtenido varias pruebas positivas, es decir, hemos podido detectar errores en la implementación, y solventarlos.*
  - ¿Hemos elaborado una versión del motor especulativo que funciona en la mayoría de las situaciones? *Sí, se ha probado el motor en multitud de escenarios posibles, y en varias aplicaciones, y ha respondido de forma satisfactoria.*
  - ¿La batería de pruebas ha sido suficientemente amplia? *Sí, aunque la fecha de entrega ha condicionado esta etapa, las pruebas realizadas han cubierto multitud de escenarios.*

- **Experimentación**. Esta etapa fue una de las más importantes, puesto que se llevó a cabo la evaluación de rendimiento del nuevo motor especulativo, tarea necesaria para la obtención de unas conclusiones adecuadas. En esta etapa no sufrimos retrasos sobre el tiempo planificado.

  Los criterios de evaluación descritos para esta etapa son los siguientes (a continuación de cada criterio se exponen los resultados obtenidos al valorar cada uno de los puntos):

  - ¿Hemos elaborado una versión del motor especulativo que funciona en la mayoría de las situaciones? *Sí, porque en los experimentos realizados, no se han detectado erratas.*

- ¿Se han obtenido conclusiones de algún tipo sobre el nuevo motor especulativo? *Sí, hemos podido establecer conclusiones sobre la nueva versión del motor especulativo.*
- ¿Se ha cumplido los objetivos del proyecto? *Sí, se ha obtenido una nueva arquitectura del motor especulativo, que solventa las limitaciones del motor original.*
- ¿Se han cumplido todos los requisitos del proyecto? *Sí, se ha revisado la lista de requisitos, sin que exista ninguno que no se cumple.*

En el cuadro 4.2 se muestra el seguimiento de la planificación.

4.4. Control de calidad

Por las características del proyecto no se han recurrido a estándares de calidad. Sin embargo no hemos querido pasar por alto un control de la calidad del proyecto y hemos decidido que todos los artefactos pasarán por un proceso de revisión por parte del alumno y los tutores asociados al Trabajo de Fin de Grado. Los defectos detectados serán sometidos a un seguimiento posterior para asegurar la solución de esos errores.

4.5. Gestión de configuraciones

La gestión de configuraciones se llevará a cabo en el servidor del grupo de investigación. Allí se almacenarán los artefactos necesarios para la consecución del proyecto. Estos incluyen: código fuente, documentación y datos. Todos ellos se incluyen en líneas base del proyecto (registros del estado de artefactos, una versión concreta). La documentación del código fuente también se incluye como documentación específica de diseño.

4.6. Presupuesto

Las estimaciones de costes son necesarias para establecer un presupuesto para el proyecto o para asignar un precio para el software de un cliente. Existen tres parámetros involucrados en el cálculo del coste total de un proyecto de desarrollo de software.

- Los costes hardware y software, incluyendo el mantenimiento.
- Los costes de viaje y capacitación.
- Los costes de esfuerzo (los costes correspondientes al pago de los ingenieros).

Tanto en este proyecto como en otros muchos, los costes dominantes son los costes de esfuerzo. Los ordenadores con potencia suficiente para desarrollar software son relativamente baratos. Por otro lado, el software necesario para el desarrollo de la mayoría del Proyecto ha sido a gratuito, dado que posee licencia libre en su mayoría[2]. Aunque haya costes de viaje, son una pequeña parte comparados con los costes de esfuerzo. Además, el uso de correo electrónico, teléfono y sitios web compartidos reducen el coste de los viajes y del tiempo hasta en un 50 %.

[2]El precio de compra del software *Microsoft Project 2010 Professional* asciende a los 1 067,00 €, sin embargo, nosotros hemos recurrido al convenio que posee la Universidad de Valladolid con la empresa que lo suministra. Esto nos ha permitido adquirirlo sin coste alguno.

Id	Nombre de tarea	Esfuerzo estimado (Horas Persona)	Esfuerzo real (Horas Persona)	Comienzo real	Fin real	Predecesoras
1	<Inicio>	0	0	lun 03/10/2011	lun 03/10/2011	
2	**Introducción a los conceptos teóricos**	**110**	**110**	**lun 03/10/2011**	**mié 02/11/2011**	**1**
3	Estudio de las características de la paralelización especulativa	50	50	lun 03/10/2011	vie 14/10/2011	
4	Estudio de las características del motor de paralelización especulativa "`SpecEngine`"	25	25	lun 17/10/2011	vie 21/10/2011	3
5	Descripción de los objetivos del proyecto	5	5	lun 24/10/2011	vie 24/10/2011	4
6	Descripción del alcance del del proyecto	5	5	lun 25/10/2011	vie 25/10/2011	5
7	Elicitación de los requisitos principales del proyecto	25	25	lun 26/10/2011	mié 02/11/2011	6
8	**Obtención de soluciones teóricas**	**450**	**540**	**jue 03/11/2011**	**mié 02/04/2012**	**2**
9.1	Obtención de la primera solución	90	140	jue 03/11/2011	mié 14/12/2011	
10.1	Análisis de la primera solución	40	50	jue 15/12/2011	mié 12/01/2012	9.1
11.1	Análisis de costes de sus operaciones	15	15	jue 13/01/2012	mar 17/01/2012	10.1
9.2	Obtención de la segunda	90	100	mié 18/01/2012	mié 15/02/2012	
10.2	Análisis de la segunda solución solución	40	40	jue 16/02/2012	mié 27/02/2012	9.2
11.2	Análisis de costes de sus operaciones	15	15	jue 28/02/2012	jue 01/03/2012	10.2
9.3	Obtención de la tercera	90	90	vie 02/03/2012	mar 27/03/2012	
10.3	Análisis de la tercera solución solución	40	60	mié 28/03/2012	jue 12/04/2012	9.3
11.3	Análisis de costes de sus operaciones	15	15	vie 13/04/2012	mar 17/04/2012	10.3
12	Elección de una solución	15	15	mié 18/04/2012	vie 20/04/2012	11.1, 11.2, 11.3
13	**Elaboración de una versión del motor especulativo que soporte aritmética de punteros**	**150**	**170**	**mar 24/04/2012**	**vie 08/06/2012**	**8**
14	**<Obtención de una implementación del nuevo motor especulativo>**	**0**	**0**	**lun 11/06/2012**	**lun 11/06/2012**	**13**
15	**Pruebas**	**25**	**50**	**lun 11/06/2012**	**vie 22/06/2012**	**14**
16	Distinto número threads	10	10	lun 11/06/2012	mar 12/06/2012	
17	Distinto tamaño de bloque	15	40	mié 13/04/2012	vie 22/06/2012	
18	**Experimentación**	**50**	**50**	**lun 25/06/2012**	**vie 06/07/2012**	**15**
19	Ejemplo sintético	15	15	lun 25/06/2012	mié 27/06/2012	
20	Menor círculo contenedor	20	20	jue 28/06/2012	mar 03/07/2012	
21	Envolvente convexo	15	15	mié 04/07/2012	vie 06/07/2012	
22	**<Fin>**	**0**	**0**	**lun 09/07/2012**	**lun 09/07/2012**	**18**
	TOTAL	**785**	**920**	**lun 03/10/2011**	**06/07/2012**	

Cuadro 4.2: Seguimiento de la planificación del proyecto.

Los costes de esfuerzo no son solo los salarios de los ingenieros que intervienen en el proyecto. Las organizaciones calculan los costes de esfuerzo en función de los costes totales, donde se tiene en cuenta el coste total para hacer funcionar la organización y dividen éste entre el número de personas productivas. Por lo tanto, los siguientes costes son parte de los costes totales.

1. Costes de proveer, aclimatar e iluminar oficinas.
2. Los costes del personal de apoyo como administrativos, secretarias, limpiadores y técnicos.
3. Los costes de redes y de comunicaciones.
4. Los costes de los recursos centralizados como las bibliotecas, los recursos recreativos, etc.
5. Los costes de seguridad social, pensiones seguros privados, etc.

Sin embargo, como en nuestro proyecto todos los costes mencionados corren a cargo de la Universidad de Valladolid, se ha decidido no tenerlos en cuenta a la hora de elaborar un presupuesto. En el cuadro 4.3 se pueden ver los costes estimados de este proyecto. Cabe mencionar que consideraremos de cuatro años el periodo de duración de un ordenador normal, y de dos años, el periodo de duración de un servidor compartido. Citamos la frase anterior porque calcularemos el precio de una máquina en el proyecto actual como el precio de compra dividido entre el número de años de duración, multiplicado por el número de años de duración del proyecto (aproximadamente uno en nuestro caso).

La estimación del precio final del software debe realizarse de forma objetiva e intentando predecir lo mejor posible el coste de desarrollo. Si el coste del proyecto se calcula dentro de un presupuesto para un cliente, entonces tendremos que decidir cómo se relacionan ambos conceptos. De forma clásica, el precio es simplemente el coste más el beneficio. Sin embargo, la relación entre el coste del proyecto y su precio no es tan simple.

Al asignar un precio al software debemos tener en cuenta los siguientes factores:

- Oportunidad de mercado: una organización podría ofertar un bajo precio debido a que desea conseguir cuota de mercado.
- Incertidumbre en la estimación de los costes: si una empresa está insegura de su coste estimado, puede incrementar su precio por encima del beneficio normal para cubrir alguna contingencia.
- Términos contractuales: un cliente puede estar dispuesto a permitir que el desarrollador retenga la propiedad del código fuente y que reutilice el código en otros proyectos. Si esto ocurriese el precio podría ser menor.
- Volatilidad de los requisitos: si es probable que los requisitos cambien, una organización puede reducir los precios para ganar un contrato.
- Salud financiera: los desarrolladores en dificultades financieras podrían bajar sus precios para obtener un contrato.

Concepto	Cantidad	Precio(€)
Herramientas de desarrollo		
GCC v4.4.3	1	0,00
gdb v7.1	1	0,00
Ubuntu Linux 11.04 - Natty Narwhal	1	0,00
VIM - Vi IMproved 7.2	1	0,00
Netbeans IDE 6.9	1	0,00
Microsoft Project 2010	1	0,00
Inkscape v0.48	1	0,00
LaTeX	1	0,00
Subtotal		0,00
Servidores (dur. estimada: dos años)		
Intel Dual Core 2.40GHz y 4GB RAM	1	500,00
Subtotal		500,00
Máquinas (dur. estimada: cuatro años)		
Acer Extensa 5635ZG 2.0GHz y 4GB RAM	1	125,00
Subtotal		125,00
Horas de trabajo		
Número de horas	785	12,50
Subtotal		9 812,50
Total		**10 437,50**

Cuadro 4.3: Costes asociados al proyecto.

Por lo tanto no existe una relación sencilla entre el precio del software que se daría al cliente y el coste de su desarrollo.

Con el deseo de obtener unos costes lo más realista posible, utilizando la técnica de estimación por analogía[3], a través de consultas a sendos ingenieros que han realizado proyectos similares, el coste final sería aproximadamente de:

Costes asociados al proyecto: 10 437,50 €.

Ganancia: 12 000 €.

TOTAL: 22 437,50 €.

[3]Estimación por analogía: esta técnica es aplicable cuando otros proyectos en el mismo dominio de aplicación se han completado. Se estima el coste de un nuevo proyecto por analogía con estos proyectos completados.

Capítulo 5

Análisis de requisitos

Aquí están especificados los requisitos, tanto funcionales como no funcionales, que debe cumplir el motor especulativo.

5.1. Requisitos funcionales

Los requisitos necesarios para que el motor paralelice especulativamente alguna aplicación son:

RF.01: el motor deberá ser capaz de paralelizar algún bucle secuencial para cualquier algoritmo realizando las modificaciones pertinentes en el código.

RF.02: para que dos iteraciones puedan ejecutarse concurrentemente en distintos procesadores, no deben existir dependencias entre los resultados calculados en la primera y los datos utilizados en la segunda. En caso contrario se producirá una violación de dependencia.

RF.03: el programador deberá clasificar todas las variables que utiliza la aplicación como datos privados o compartidos.

RF.04: el programador deberá identificar entre las variables compartidas, aquellas que pueden producir violaciones de dependencia. Estas variables serán identificadas como especulativas.

- **Observaciones**: una variable producirá una violación de dependencia cuando su valor se modifica en una iteración determinada y un sucesor ha consumido con anterioridad un valor antiguo de esa variable.

RF.05: el programador deberá indicar el número de threads que quiere utilizar, el tamaño del bucle a paralelizar y el número de iteraciones a ejecutar en cada bloque.

RF.06: el número de threads a utilizar deberá ser menor o igual al número de procesadores disponibles sobre la máquina en la que se trabaja.

RF.07: cada vez que aparezca una lectura sobre una variable especulativa, el programador deberá sustituirlo por una llamada al motor, con el fin de realizar una lectura especulativa.

RF.08: cada vez que aparezca una escritura sobre una variable especulativa, el programador deberá sustituirlo por una llamada al motor, con el fin de realizar una escritura especulativa.

RF.09: una vez que un bloque de iteraciones haya sido ejecutado, el programador deberá solicitar al motor la consolidación de los datos.

RF.10: una vez que un bloque de iteraciones haya sido ejecutado, el motor deberá asignar un nuevo bloque de iteraciones, si lo hubiese.

RF.11: los resultados obtenidos por la ejecución especulativa serán iguales a los que proporcione la versión especulativa, con independencia del número de procesadores.

RF.12: el nuevo motor será capaz de paralelizar especulativamente todas las aplicaciones que paralelizaban los motores precedentes.

RF.13: el nuevo motor deberá paralelizar especulativamente aplicaciones que utilicen aritmética de punteros.

RF.14: la nueva arquitectura del motor deberá posibilitar la paralelización especulativa de aplicaciones que utilicen variables con tipos diferentes.

RF.15: el nuevo motor deberá permitir especular sobre variables aisladas.

5.2. Requisitos no funcionales

5.2.1. Rendimientos

RNF.01: el tiempo de respuesta después de paralelizar una aplicación dependerá del número de violaciones de dependencia que se produzcan, pero en general, deberá de ser menor o igual (dependiendo del número de procesadores que se empleen) al de la aplicación secuencial.

RNF.02: los recursos de memoria empleados aumentará proporcionalmente con el número de hilos empleados.

5.2.2. Restricciones de diseño

RNF.03: las versiones previas del motor que se utilizarán como base serán las implementadas por el Grupo Trasgo.

RNF.04: el nuevo motor deberá ser implementado en el lenguaje C.

RNF.05: para desarrollar la aplicación sólo se utilizarán herramientas de GNU.

RNF.06: el compilador de C será GCC v4.4.3 en Ubuntu Linux.

RNF.07: los experimentos serán ejecutados sobre el servidor *nodoyuna*, un Intel Dual Core equipado con un procesador quad-core Intel Core2 Q6600 a 2.40GHz, 4096 KB de memoria caché y 2.7GB de RAM.

RNF.08: para la medición de rendimientos se emplearán las funciones estándar de UNIX.

RNF.09: las imágenes serán diseñadas mediante el editor de gráficos vectoriales Inkscape 0.48.

RNF.10: la documentación relativa al proyecto deberá desarrollarse mediante LaTeX.

5.2.3. Interfaces

RNF.11: tanto el motor como el programador deberán utilizar la API de OpenMP para el desarrollo de aplicaciones paralelas.

Capítulo 6

Uso de matrices de versión de datos y matrices de punteros para cada thread

A fin de lograr una versión del motor especulativo lo más eficiente posible, abordaremos varias implementaciones posibles. A lo largo de este capítulo estudiaremos la primera de ellas, una solución basada en realizar los menos cambios posibles al motor existente, y reutilizar lo posible.

6.1. Bases de la solución

A continuación vamos a citar los elementos de que constará esta idea:

- **Matriz de versiones:** en la nueva implementación del motor, habrá una matriz que almacenará la versión actual de los datos. Esta matriz existía en las versiones previas del motor. Para elaborar esta propuesta se ha decidido no crear una matriz bidimensional nueva, sino que reutilizaremos la matriz actual que almacena datos (version) de tal forma que esta estructura actuará como si fuese la primera columna de una matriz bidimensional, e implementaremos otra estructura que almacene las direcciones a memoria, PM (Pointer Matrix), como segunda columna.
- **Matriz de punteros:** esta matriz actuará como segunda columna de una matriz bidimensional. En ella almacenaremos las direcciones de los datos contenidos en la matriz de versiones.

De esta manera el acceso a la posición de la estructura se realizará de forma similar a el acceso a la matriz de versiones, puesto que serán matrices similares.

Cabe mencionar que las matrices de versiones y punteros serán diferentes para cada thread, dado que cada uno almacenará solamente los punteros y datos que necesite, es decir, si el thread N tiene en su posición 1 el puntero Q, el thread N-1 puede contener en la posición 1 cualquier otro puntero, no necesariamente el Q. Esto se diferencia de la anterior implementación del motor, donde el número de datos era similar en todos los threads, y el valor de una

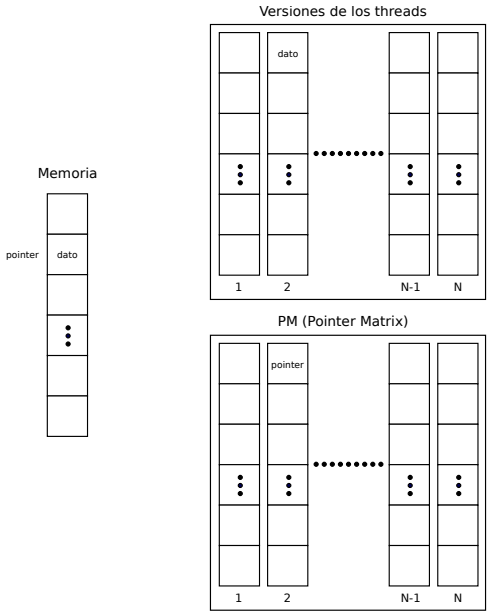

Figura 6.1: Solución 1: estructuras principales.

posición de matriz de versiones de un thread se refería al mismo dato que la de los demás threads.

En el motor ya no existirá versión principal de los datos puesto que ya no tiene sentido, los datos se modificarán o consultarán directamente de la memoria. Las estructuras principales a utilizar se pueden ver en la figura 6.1.

Para implementar la nueva modificación, habría que modificar el código actual del motor, sin embargo, para realizar las nuevas operaciones requeridas nos basaremos en las ya existentes, realizando donde sea necesario los cambios pertinentes.

Otra consideración a la hora de la implementación sería la reutilización de la matriz que contenía la copia principal de los datos (`ref`), de tal manera que será nuestra matriz de punteros (`PM`). Así, cada vez que un thread consolide sus datos, modificará el valor directamente de memoria, y cuando lo consulte lo hará directamente de memoria.

Además para realizar la implementación se tendría en cuenta que para optimizar en C las búsquedas lineales, se debe trasponer la matriz de punteros PM. Esto es debido a que en C, las matrices se guardan por filas, no por columnas. Con esta modificación los accesos a memoria serán contiguos, no habrá que dar saltos sucesivos para comprobar posiciones sucesivas.

6.2. Modificación en la operación de lectura

Veamos el código explicativo de la operación mencionada:

```
// Sea tid el thread en cuestion,
// pointer el valor del puntero
// max será la posición máxima del vector PM, es decir, la primera donde no haya puntero
```

```
k=0;
max = -1;
th=tid;

while (th>=0){
  if (PM[k][th] == NoPointer){ // si el valor es igual a que no haya puntero
    th--; // disminuimos el thread que consultamos
    if (th == tid){ // si es el thread que intenta leer;
      max = k; // asignamos su posición máxima
      k=0;
    }
  }else if (PM[k][th] == pointer){ // si el valor es el buscado
    if (AM[k][th] == Mod || AM[k][th] == ModExpLd){
      forwarded = TRUE;
      break;
    }
  }else
    k++;
}

PM[max][tid] = pointer; // se asigna a la PM del puntero que quiere leerlo
if (forwarded)
  version[max][tid] = version[k][th]; // se cambia el valor de su versión
else
  version[max][tid] = *pointer; // lo cargamos de memoria hacia la version del thread

AM[max][tid] = ExpLd;
PM[max+1][tid] = NoPointer; // se asigna a la PM del puntero que quiere leerlo

lvalue = version[max][tid];
```

Veremos un ejemplo con imágenes. Supongamos que tenemos inicialmente los valores vistos en la figura 6.2(a) en las estructuras del motor especulativo. Imaginemos que en este momento el thread número dos procesa la siguiente instrucción:

```
A = *p;
```

El thread quiere obtener el valor de una posición de memoria. Supongamos que `p` vale 4 000 y que `*p = dato`. ¿Cómo se llevaría a cabo la operación de lectura?

1. El thread dos buscaría en su PM el valor de p: 4 000. Véase figura 6.2(b).
2. Como no se ha encontrado el valor de la posición de memoria (sólo contiene el valor 1 000), significa que este thread no ha utilizado anteriormente este puntero. Por tanto, busca en la matriz PM de los threads anteriores para saber si alguno ha utilizado este valor. En este caso busca en el thread uno, pero este tampoco contiene el valor 4 000, contiene los valores 3 500, 2 500 y 1 000. Descrito en la figura 6.2(c).
3. Como los threads predecesores no contienen el puntero en su PM, el thread dos carga el valor de ese puntero directamente de memoria, agrega a su PM la posición de memoria del dato, en este caso `4 000`, y por último en su matriz de versiones, en la misma posición en que se ha almacenado el puntero en la matriz PM, se guarda el valor del dato, en este caso `dato`. Esto se puede ver en la figura 6.2(d).

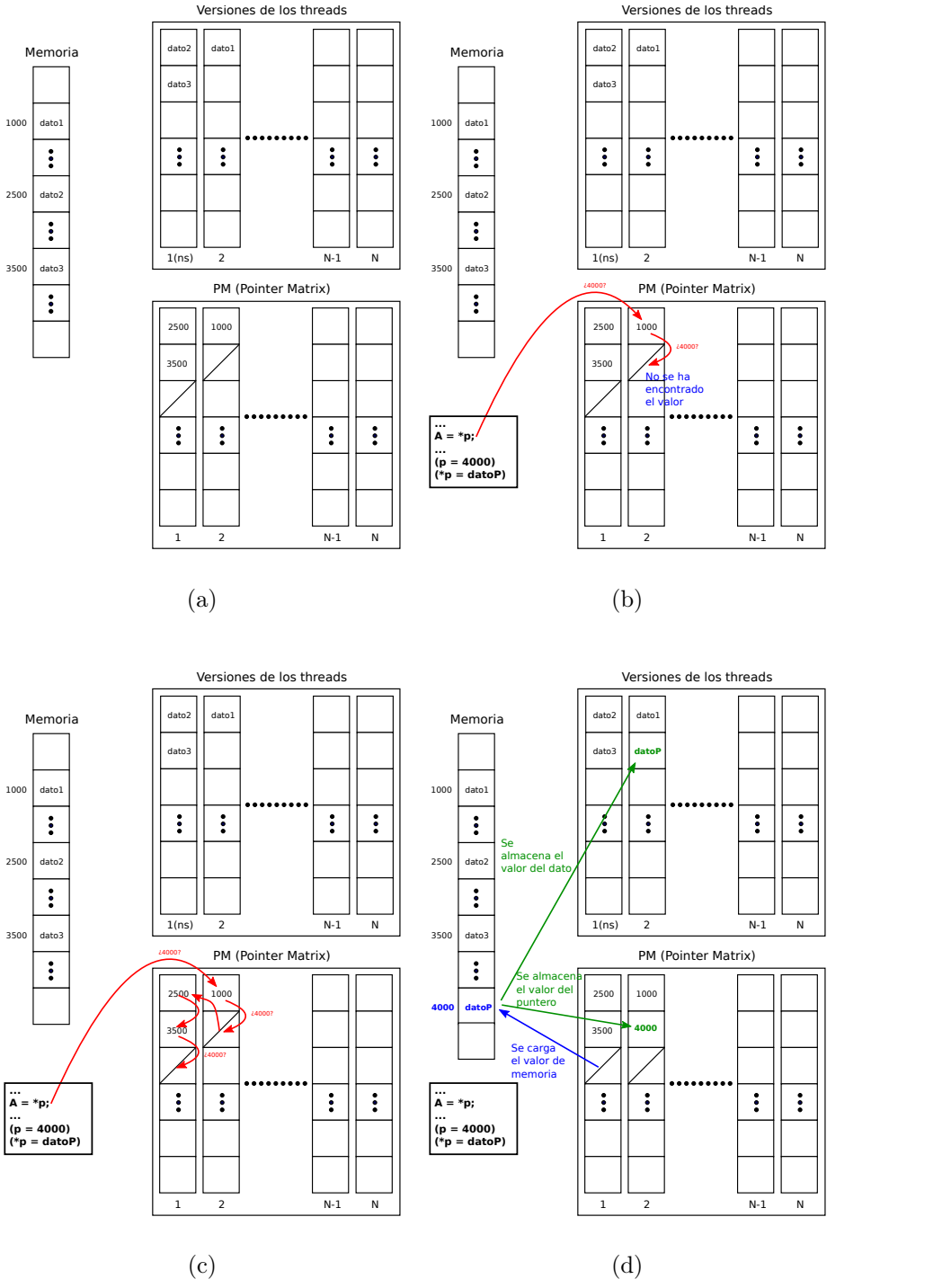

Figura 6.2: Solución 1: operación de lectura. (a) Valores iniciales de nuestro ejemplo. (b) El thread busca el valor del puntero en su matriz PM. (c) El thread busca el valor en los threads predecesores. (d) Se carga el valor de memoria (azul), se deposita el valor del puntero en la matriz de versiones (verde) y se deposita el valor del puntero en la matriz PM (rosa).

6.3. Modificación en la operación de escritura

Veamos el código explicativo de la operación mencionada:

```
// Sea tid el thread en cuestion,
// pointer el valor del puntero
// valor es el valor a escribir
// max será la posición máxima del vector PM, es decir, la primera donde no haya puntero

max = -1;

for (i = 0; i < number_of_variables; i++){
  if (PM[i][tid] == pointer){
    pos = i;
    break;
  }else if (PM[i][tid] == NoPointer){
    max = i;
    break;
  }
}

if (max != -1){
  PM[max][tid] = pointer;
  PM[max+1][tid] = NoPointer;
  AM[max][tid] = NotAcc;
  pos = max;
  max++;
}

if (AM[pos][tid] == NotAcc){
  AM[pos][tid] = Mod;
  IM[++tail][tid]][tid] = pos;
}
if (AM[pos][tid] == ExpLd)
  AM[pos][tid] = ExpLdMod;

version[pos][tid] = valor;

k=0;
th=tid+1;

while (th<N){
  if (PM[k][th] == NoPointer){ // si el valor es igual a que no haya puntero
    th++; // disminuimos el thread que consultamos
    k=0;
  }
  else if (PM[k][th] == pointer){ // si el valor es el buscado
    if (AM[k][th] == Mod)
      break;
    else if (AM[k][th] != NotAcc){
      squash(th);
      break;
```

```
    }
  }else
    k++;
}
```

Para este caso también veremos un ejemplo con imágenes. Supongamos que tenemos inicialmente los valores vistos en la figura 6.3(a) en las estructuras del motor especulativo. Imaginemos que en este momento el thread número dos procesa la siguiente instrucción:

```
*p = datoP;
```

El thread procesa una instrucción de escritura en una posición de memoria. Supongamos que `p = 4 000`. ¿Cómo se llevaría a cabo la operación de escritura?

1. El thread dos buscaría en su PM el valor de `p`: 4 000. Véase figura 6.3(b).
2. Como no se ha encontrado el valor de la posición de memoria (sólo contiene el valor 1 000), significa que este thread no ha utilizado anteriormente este puntero. Por tanto, añade el valor del puntero a la matriz PM, y también el contenido de la posición de memoria apuntada por el puntero se agrega a la matriz de versiones del thread. Descrito en la figura 6.3(c).
3. Por último, se comprueba que los threads posteriores no hayan utilizado el valor del puntero 4 000, para ello se recorren uno a uno las matrices PM de los threads sucesores. Se puede ver una descripción en la figura 6.3(d). Si alguno lo hubiese utilizado se marcaría como `SQUASHED`, pero en el caso descrito en las imágenes no se producirá esta situación.

6.4. Modificación en la operación de consolidación

Veamos el código explicativo de la operación mencionada:

```
// Sea tid el thread en cuestion,
// I el valor del índice de la matriz

1.  #pragma critical
2.  if (window[tid] != SQUASHED) {
3.    if (tid == non_spec) {
4.      window[tid]=DONE;
5.      for (i=non_spec; i<=most_spec; i++) {
6.        if (window[i] == DONE &&
              window[i+1] != DONE)
7.          { last=i; break; }
8.      }
9.      for (j=non_spec; j<=last; j++) {
           k = 0;
           while (PM[k][j] != NoPointer){
             *PM[k][j] = version[k][j];
             PM[k][j] = NoPointer;
             AM[k][j] = NotAcc;
```

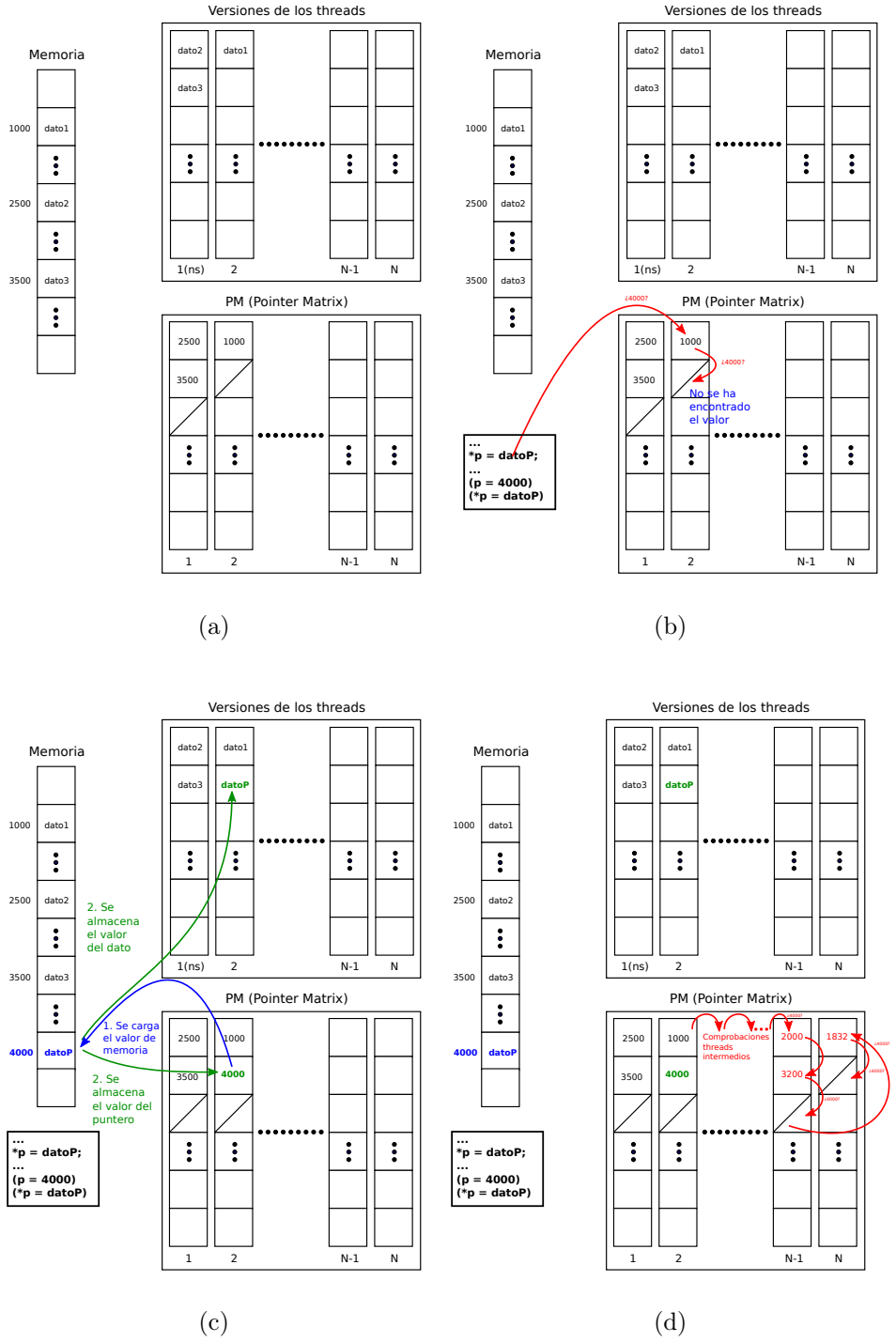

Figura 6.3: Solución 1: operación de escritura. (a) Valores iniciales de nuestro ejemplo. (b) El thread busca el valor del puntero en su matriz PM. (c) El thread añade el valor del puntero a su matriz PM y el contenido del mismo a su matriz de versiones. (d) El thread que escribe el nuevo valor comprueba si alguno de los threads sucesores ha utilizado dicho valor, y por tanto, hay que marcarlo para su re-ejecución.

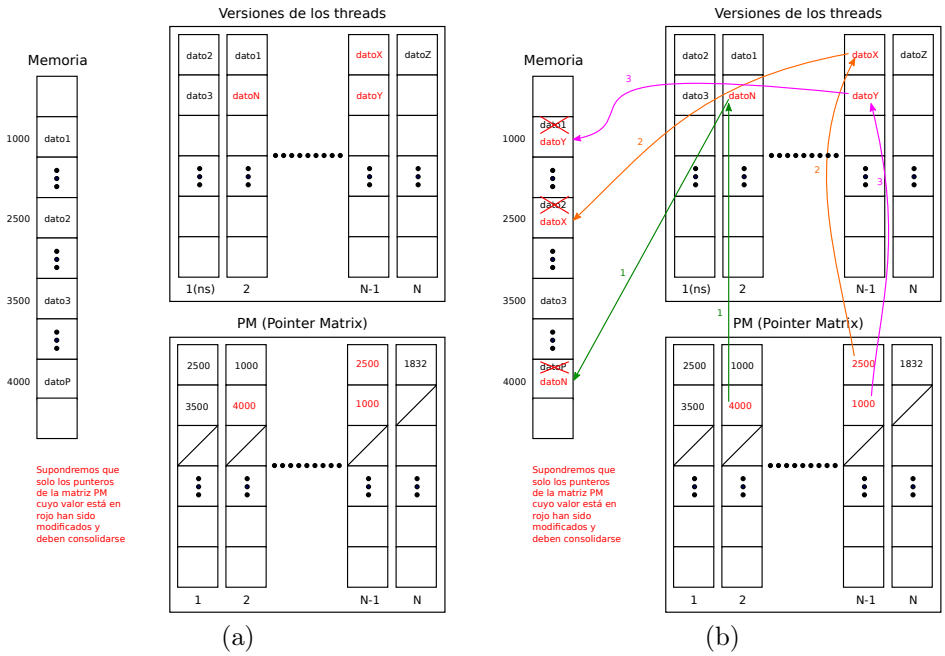

Figura 6.4: Solución 1: operación de consolidación. (a) Valores iniciales de nuestro ejemplo. (b) Consolidación de los datos en memoria en orden. Los números indican el orden de consolidación.

```
            k++;
          }
14. #pragma memory fence
15.         window[j]=FREE;
16.     }
17.     non_spec=last+1;
18.   }
19.   else
20.     window[tid]=DONE;
21. }
22. else do_squash();
23. #pragma end critical
24. while (window[most_spec+1] != FREE) {}
25. #pragma critical
26. for (j=1; j<=tail[most_spec+1]; j++)
27.   AM[IM[j][most_spec+1][most_spec+1]=NotAcc;
28. #pragma memory fence
29. tail[most_spec+1]=0;
30. window[most_spec+1]=RUN;
31. most_spec++;
32. #pragma end critical
```

Para el caso de la consolidación de resultados también veremos un ejemplo con imágenes. Supongamos que tenemos inicialmente los valores vistos en la figura 6.4(a) en las estructuras del motor especulativo. Supongamos que los únicos valores que se deben consolidar son los

marcados en rojo en la figura recién mencionada, consideraremos que los otros simplemente se han leído.

Tras finalizar la ejecución de sus instrucciones, ¿Cómo se llevaría a cabo la operación de consolidación?

1. La consolidación se realizará en orden, es decir, el thread no especulativo buscará entre sus valores si alguno ha sido actualizado consultando su matriz AM. En caso de que alguno haya sido actualizado, escribe su contenido en memoria. Una vez finalizada su consolidación pasa a estado `FREE` y el thread no especulativo pasa a ser el sucesor, que si ha finalizado su ejecución, también se consolidará. En nuestro caso consideraremos que el threads no especulativo es el uno, y que todos los threads han finalizado su ejecución. Los resultados se pueden ver en la figura 6.4(b).

6.5. Análisis de los costes

Aquí se realizará un cálculo de las operaciones que se sucederán en las ejecuciones del motor tomando como escenario el peor caso posible. Como datos de referencia del estudio, se tomarán `T` threads y `Q` punteros.

6.5.1. Coste de la operación de lectura

Situación de partida: en este escenario el thread más especulativo quiere leer un dato de memoria, por ello utiliza la operación del motor specload_pointer(). Los `T-1` primeros threads tienen `Q` punteros, y el thread `T` tiene `Q-1` punteros en la matriz PM.

Primera operación: el thread número `T` busca en su matriz de punteros (PM) el valor de la posición de memoria del dato a consultar. Tras llegar al último elemento, el `Q`, sin haber encontrado el valor, consulta en la PM del thread anterior. Este proceso se repetiría sin éxito hasta alcanzar el último elemento del thread no especulativo, el primero. En este momento tenemos `T*Q-1` comparaciones.

Segunda operación: como no se ha encontrado el valor, hay que buscar su valor en memoria. Esta operación es inmediata, un acceso a memoria y no se tendrá en cuenta.

Tercera operación: en este punto se debe almacenar el valor de la posición de memoria en la matriz de punteros del thread `T` y el valor del dato en la matriz de versión, ambas en la misma posición. Como se puede comprobar en este caso realizamos dos operaciones más, si conocemos el primer valor vacío de nuestras matrices, o `(Q-1) + 2` operaciones si no lo conocemos. Tomaremos en este caso el peor valor posible.

Conclusión: el coste total será de: $CT \in O((T*Q)-1+(Q-1)+2) \in O(T*(Q+1)) \in O(T*Q)$ operaciones.

6.5.2. Coste de la operación de escritura

Situación de partida: en este escenario el thread menos especulativo quiere escribir un dato en memoria, por ello utiliza la operación del motor specstore_pointer(). Los `T` threads tienen `Q` punteros en la matriz PM, salvo el no especulativo que tiene `Q-1`.

Primera operación: el thread no especulativo busca en su matriz PM si existe el puntero donde escribir, como tiene `Q-1` punteros, realizará `Q-1` comparaciones. Al no existir dicho

valor, lo agrega en la última posición, y añade en su matriz de versiones el valor a escribir. Esto suma 2 operaciones más.

Segunda operación: en este punto debemos comprobar si el dato ha sido utilizado por algún thread posterior, para detectar posibles squashes. Supongamos que ningún thread ha utilizado ese dato, esto nos lleva al peor caso porque necesitaremos `(T-1) * Q` comparaciones, una por cada posición de la matriz PM del thread.

Conclusión: el coste total será de: $CT \in O((Q-1)+2+(T-1)*Q) \in O(Q+1+T*Q-Q) \in O(T*Q+1) \in O(T*Q)$ operaciones.

6.5.3. Coste de la operación de consolidación

Situación de partida: en este escenario el thread más especulativo finaliza su ejecución y quiere proceder a consolidar sus datos en memoria, por lo que utiliza la operación del motor correspondiente. Los `T` threads tienen `Q` punteros en la matriz PM.

Primera operación: el thread más especulativo debe esperar a que los `T-1` threads anteriores finalicen, esto produce un tiempo de espera, T_1. Imaginemos que durante la ejecución, el thread menos especulativo provoca que todos los demás invaliden su ejecución por squash. Cuando el thread no especulativo finaliza, hay que realizar `Q` operaciones.

Segunda operación: cuando finaliza el thread no especulativo, pasamos a tener otro thread no especulativo, por lo que hay que esperar que este finalice, por ello sumamos un tiempo T_2. Cuando por fin finaliza, también provoca squash en todos los demás y se consolida, realizando `Q` operaciones más. Estas operaciones se repetirán hasta consolidar el thread inicialmente más especulativo, el que queríamos consolidar al principio.

Conclusión: el coste total será proporcional a la suma del tiempo de espera de las operaciones que requieran consolidación, esto es, $O(T_1+Q+T_2+Q+...+T_T+Q) \in O(Q*T)$ y a los tiempos de espera de las operaciones que no requieran un commit, es decir, $(O(\sum_{i=0}^{T} T_i)$, esto es, $O(CT) \in O(Q*T+\sum_{i=0}^{T} T_i)$.

Capítulo 7

Uso de una matriz global de punteros y matrices de versión de datos para cada thread

Realizaremos un estudio de las bases de esta solución, y de las operaciones que se llevarían a cabo. La solución en que se centra este capítulo se basa en la utilización de una matriz principal a la que accederán todos los threads.

7.1. Bases de la solución

En este momento, nos parece interesante investigar sobre las posibilidades que da la segunda de las ideas, por ello citaremos los elementos principales de que constaría el motor:

- **Matriz principal:** esta estructura contendrá los datos de los punteros de cada thread, y éstos le consultarán como referencia.
- **Matriz de versiones:** al igual que en la solución anterior, existirá una matriz de versiones que almacenará la copia más reciente del dato para cada thread. Cada versión tendrá capacidad para almacenar todos los datos, es decir tendrá las mismas filas que la matriz principal.

Como ventaja de esta idea podemos destacar que será mucho más rápida que la anterior, y que no hace falta implementar la matriz de punteros PM.

Por otro lado podemos destacar que necesitaremos `N+1` estructuras (siendo `N` el número de threads) de un tamaño elevado, mayor que el que se necesitaría en el caso anterior porque cada thread contendrá en su matriz de versiones tantas posiciones como punteros haya en el bucle a paralelizar especulativamente.

La nueva matriz será denominada MM (Main Matrix), y si el número de punteros del bucle es `Q`, tendrá un tamaño de `Q` filas por 2 columnas. La primera columna será la posición de memoria, y la segunda será el valor de esa posición. Para entender mejor esta idea veamos las estructuras principales que utilizará en la figura 7.1.

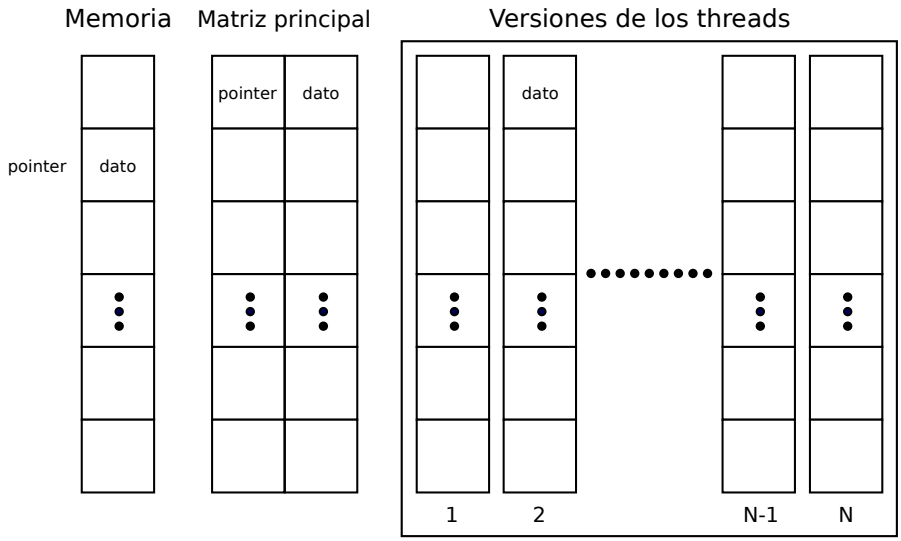

Figura 7.1: Solución 2: estructuras principales.

7.2. ¿Cual es la principal ventaja de esta implementación?

El acceso a datos es directo, es decir, no habrá que realizar búsquedas constantes, simplemente consultar en la matriz principal que posición ocupa el puntero a consultar, si no existe cargarlo de memoria, y si existe copiarlo a la matriz de versiones donde corresponda.

7.3. Modificación en la operación de lectura

Veamos el código explicativo de la operación mencionada:

```
// Sea tid el thread en cuestion,
// pointer el valor del puntero
// max será la primera posición de la matriz MM donde no haya puntero
// Q será el número de filas de la matriz MM

k=0;
max = -1; // si el puntero no existe devolverá la posición donde se agrega
pos = -1; // si el puntero existe devuelve su posición

while (max == -1 || pos == -1 || k < Q){
  if (MM[k][0] == NoPointer){ // si el valor es igual a que no haya puntero
    MM[k][0] = puntero;
    MM[k][1] = *puntero;
    MM[k+1][0] = NoPointer;
    max = k+1;
    version[k][tid] = MM[k][1];
  }
  else if (MM[k][0] == pointer){ // si el valor es el buscado
    pos = k;
```

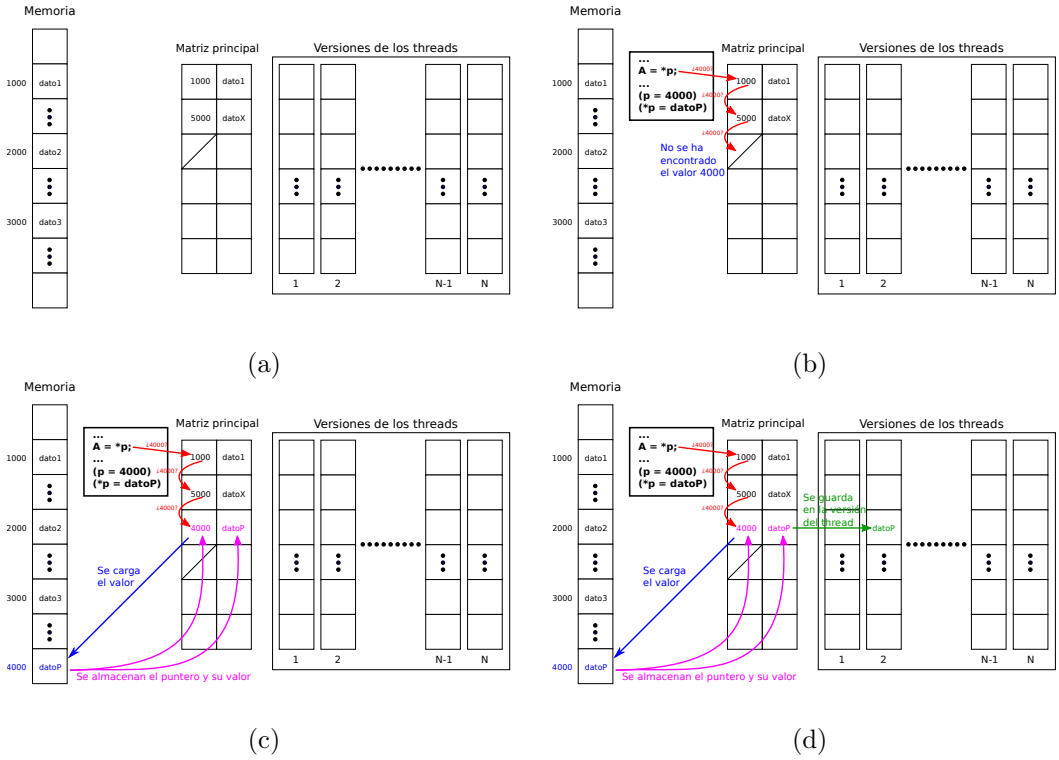

Figura 7.2: Solución 2: operación de lectura. (a) Valores iniciales de nuestro ejemplo. (b) El thread busca el valor del puntero en la matriz MM. (c) Se carga el valor de memoria, y se almacena en la primera posición vacía de la matriz MM. (d) Se almacena el valor en la versión del thread.

```
    AM[pos][tid] = ExpLd;
    for (i = tid-1; i >= non_spec; i--){
      if (AM[pos][i] == Mod || AM[pos][i] == ModExpLd){
        version[pos][tid] = version[pos][i];
        forwarded = TRUE;
        break;
      }
    }
    if (!forwarded)
      version[pos][tid] = MM[pos][1];
  }
  else
    k++;
}
lvalue = version[pos][tid];
```

Veamos un ejemplo con imágenes. Supongamos que tenemos inicialmente los valores vistos en la figura 7.2(a) en las estructuras del motor especulativo. Imaginemos que en este momento el thread número dos procesa la siguiente instrucción:

```
A = *p;
```

El thread quiere obtener el valor de una posición de memoria. Supongamos que `p` vale 4 000 y que `*p = dp`. ¿Cómo se llevaría a cabo la operación de lectura?

1. Se busca en la matriz principal MM el valor de `p`: 4 000. Véase figura 7.2(b).
2. Como no se ha encontrado el valor de la posición de memoria (sólo contiene los valores 1 000 y 5 000), significa que no se ha utilizado anteriormente este puntero. Por tanto, se carga de memoria el valor de esa posición y se almacena en la primera posición vacía de la matriz MM. Esto se describe en la figura 7.2(c).
3. Hecho esto, se cambia el valor de la versión del thread dos por el valor del dato, como se aprecia en la figura 7.2(d).

7.4. Modificación en la operación de escritura

Veamos el código explicativo de la operación mencionada:

```
// Sea tid el thread en cuestion,
// pointer el valor del puntero
// max será la primera posición de la matriz MM donde no haya puntero
// Q será el número de filas de la matriz MM
// valor es el valor a escribir

k=0;
max = -1; // si el puntero no existe devolverá la posición donde se agrega
pos = -1; // si el puntero existe devuelve su posición

while (max == -1 || pos == -1 || k < Q){
  if (MM[k][0] == NoPointer){ // si el valor es igual a que no haya puntero
    MM[k][0] = puntero;
    MM[k][1] = *puntero;
    MM[k+1][0] = NoPointer;
    max = k+1;
    version[k][tid] = valor;
  }
  else if (MM[k][0] == pointer){ // si el valor es el buscado
    pos = k;
    if (AM[pos][tid] == NotAcc)
      AM[pos][tid] = Mod;
    if (AM[pos][tid] == ExpLd)
      AM[pos][tid] = ModExpLd;
    for (i = tid+1; i <= most_spec; i++){
      if (AM[pos][i] == Mod)
        break;
      else if (AM[pos][i] != NotAcc){
        squash(i);
        break;
      }
    }
  }
```

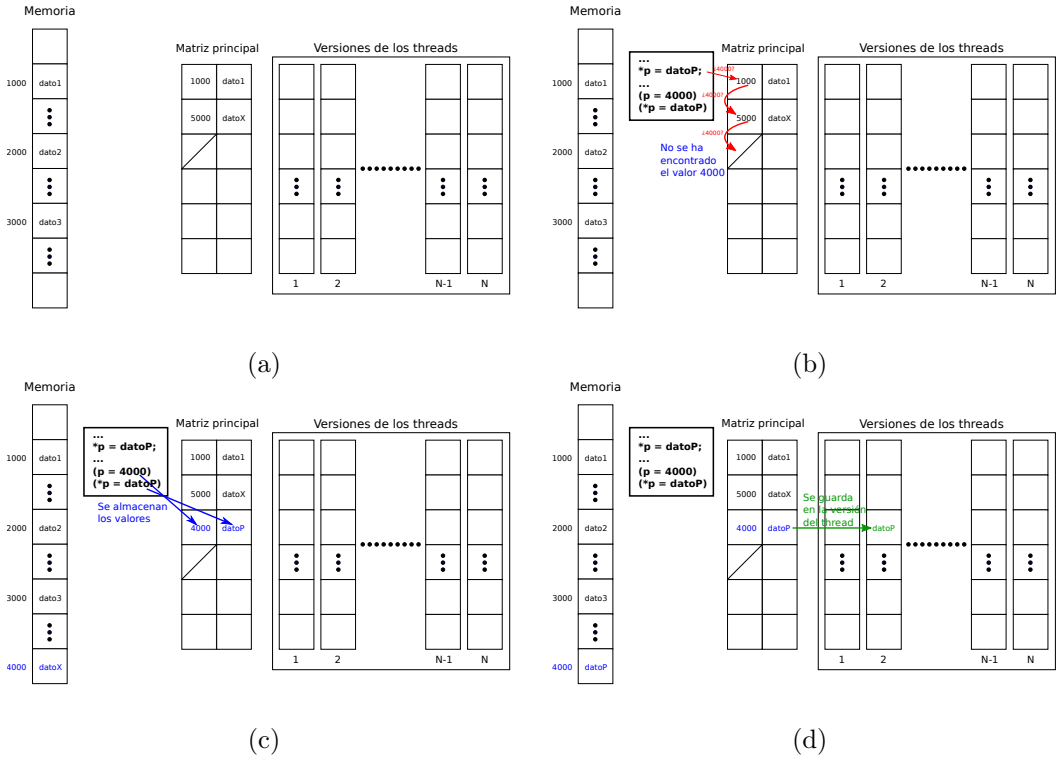

Figura 7.3: Solución 2: operación de escritura. (a) Valores iniciales de nuestro ejemplo. (b) Se busca el valor del puntero en la matriz MM. (c) Se agrega a la matriz MM el valor del puntero junto al dato que se desea escribir en esa posición de memoria. (d) Se guarda el valor a escribir en la posición correspondiente del thread que ejecutó la instrucción de escritura.

```
  else
    k++;
}
```

En este punto vamos a examinar un ejemplo con imágenes. Supongamos que tenemos inicialmente los valores vistos en la figura 7.3(a) en las estructuras del motor especulativo. Imaginemos que en este momento el thread número dos procesa la siguiente instrucción:

```
*p = datoP;
```

El thread procesa una instrucción de escritura en una posición de memoria. Supongamos que `p` vale 4 000. ¿Cómo se llevaría a cabo la operación de escritura?

1. Sería necesario buscar en la matriz principal(MM) el valor de `p`: 4 000. Véase figura 7.3(b).

2. Como no se ha encontrado el valor de la posición de memoria (sólo contiene los valores 1 000 y 5 000), significa que este puntero no ha sido utilizado anteriormente. Por tanto, agregamos el puntero en la posición correspondiente, junto al dato que se desea escribir en esa posición, como se muestra en la figura 7.3(c).

3. Por último, se almacena el valor del dato en la versión del thread que ejecutó la instrucción. Se puede ver una descripción en la figura 7.3(d).

7.5. Modificación en la operación de consolidación

Para esta solución habrá que hacer dos operaciones de consolidación, una cada vez que un thread finaliza, hacia la matriz principal MM. La otra operación de consolidación se realizará cuando finalicen todos los threads, y volcará los datos de MM hacia la memoria.

7.5.1. Commit desde threads a MM

```
// Sea tid el thread en cuestion,
// pointer el valor del puntero
// max será la primera posición de la matriz MM donde no haya puntero
// Q será el número de filas de la matriz MM
// valor es el valor a escribir

1.  #pragma critical
2.  if (window[tid] != SQUASHED) {
3.    if (tid == non_spec) {
4.      window[tid]=DONE;
5.      for (i=non_spec; i<=most_spec; i++) {
6.        if (window[i] == DONE &&
              window[i+1] != DONE)
7.          { last=i; break; }
8.      }
9.      for (j=non_spec; j<=last; j++) {
           for(k = 0; k < Q; k++){
             if (PM[k][j] != NoPointer){
               MM[k][1] = version[k][j];
               AM[k][j] = NotAcc;
             }
           }
14. #pragma memory fence
15.        window[j]=FREE;
16.     }
17.     non_spec=last+1;
18.   }
19.   else
20.     window[tid]=DONE;
21. }
22. else do_squash();
23. #pragma end critical
24. while (window[most_spec+1] != FREE) {}
25. #pragma critical
26. for (j=1; j<=tail[most_spec+1]; j++)
27.   AM[IM[j][most_spec+1][most_spec+1]=NotAcc;
28. #pragma memory fence
29. tail[most_spec+1]=0;
30. window[most_spec+1]=RUN;
```

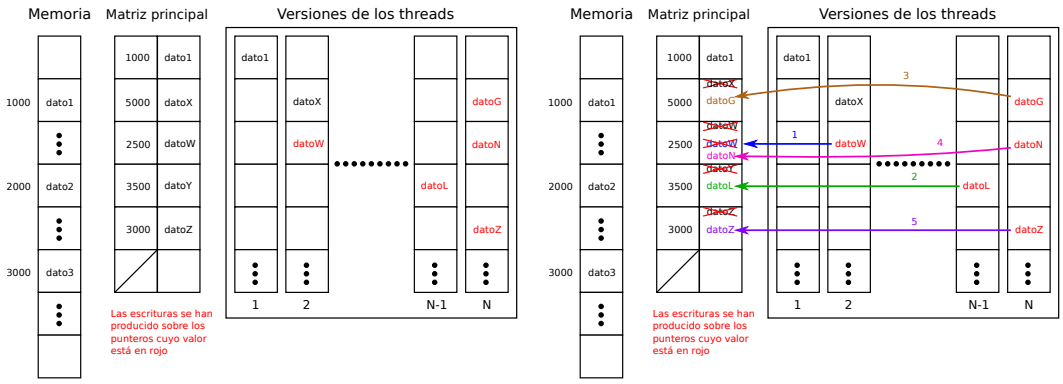

Figura 7.4: Solución 2: operación de consolidación. (a) Valores iniciales de nuestro ejemplo. (b) Consolidación de los datos en memoria en orden. Los números indican el orden de consolidación.

```
31. most_spec++;
32. #pragma end critical
```

Para el caso de la consolidación de resultados también veremos un ejemplo con imágenes. Supongamos que tenemos inicialmente los valores vistos en la figura 7.4(a) en las estructuras del motor especulativo. Supongamos que los únicos valores que se deben consolidar son los marcados en rojo en la figura recién mencionada, consideraremos que los otros simplemente se han leído.

Tras finalizar la ejecución de sus instrucciones, ¿Cómo se llevaría a cabo la operación de consolidación?

1. La consolidación se realizará en orden, es decir, el thread no especulativo buscará entre sus valores si alguno ha sido actualizado consultando su matriz AM. En caso de que alguno haya sido actualizado, escribe su contenido en memoria. Una vez finalizada su consolidación pasa a estado `FREE` y el thread no especulativo pasa a ser el sucesor, que si ha finalizado su ejecución, también se consolidará. En nuestro caso consideraremos que el threads no especulativo es el uno, y que todos los threads han finalizado su ejecución. Los resultados se pueden ver en la figura 7.4(b). Como se puede comprobar, si los valores han sido escritos, aunque el dato del puntero coincida, se sobreescribe, como se puede comprobar en la consolidación de los datos con flecha azul y morada de la figura 7.4(b).

7.5.2. Commit desde MM hacia memoria

A continuación se puede ver el código `C` que implementa esta operación.

```
// Q será el número de filas de la matriz MM

for (i=0; i<Q; i++){
  if (MM[i][0] == NoPointer)
    break;
```

```
  *MM[i][0] = MM[i][1];
}
```

Este caso es tan sencillo que no se mostrará ejemplo alguno.

7.6. Análisis de los costes

Aquí se realizará un cálculo de las operaciones que se sucederán en las ejecuciones del motor tomando como escenario el peor caso posible. Como datos de referencia del estudio, se tomarán `T` threads y `Q` punteros en la matriz principal.

7.6.1. Coste de la operación de lectura

Existen dos posibilidades a la hora de medir el coste de estas operaciones, a saber, si el dato ha sido, o no, leído anteriormente, es decir, si está en la matriz principal, o no.

1. **Dato disponible en MM:**

   Situación de partida: en este escenario el thread más especulativo quiere leer un dato de memoria, por ello utiliza la operación del motor specload_pointer().

   Primera operación: el thread número `T` busca en la matriz principal (MM) el valor de la posición de memoria del dato a consultar. Tras llegar al último elemento, el `Q`, encuentra el valor en este último elemento, lo cual ha supuesto `Q` comparaciones.

   Segunda operación: nuestro thread carga ese valor a la posición de su vector de versiones, y busca en los threads anteriores si alguno ha modificado ese valor anteriormente, agregando al coste `T` comparaciones más.

   Tercera operación: ningún thread ha actualizado ese valor, por tanto, lo carga de la matriz principal, añadiendo una operación más.

   Conclusión : el coste total será de: $CT \in O(Q+1+T+1) \in O(Q+T)$ operaciones.

2. **Dato no disponible en MM:**

   Situación de partida: en este escenario el thread más especulativo quiere leer un dato de memoria, por ello utiliza la operación del motor specload_pointer().

   Primera operación: el thread número `T` busca en la matriz principal (MM) el valor de la posición de memoria del dato a consultar. Tras llegar al último elemento, el `Q`, el valor no ha sido encontrado, sin embargo ya se han producido `Q` comparaciones.

   Segunda operación: se carga ese valor de memoria a la posición `Q + 1` de MM.

   Tercera operación: el valor obtenido se copia a la posición `Q + 1` del vector de versiones del thread más especulativo.

   Conclusión : el coste total será de: $CT \in O(Q+1+1) \in O(Q)$ operaciones.

7.6.2. Coste de la operación de escritura

al igual que en la operación de lectura, hay dos posibilidades a la hora de medir el coste de estas operaciones, a saber, si el dato ha sido, o no, utilizado anteriormente, es decir, si está en la matriz principal, o no.

1. **Dato disponible en MM:**

   Situación de partida: en este escenario el thread menos especulativo quiere escribir un dato en memoria, por ello utiliza la operación del motor specstore_pointer().

   Primera operación: el thread número uno busca en la matriz principal (MM) el valor de la posición de memoria del dato a almacenar. Tras llegar al último elemento, el `Q`, encuentra el valor en este último elemento, lo cual ha supuesto `Q` comparaciones.

   Segunda operación: en este punto debemos comprobar si el dato ha sido utilizado por algún thread posterior, para detectar posibles squashes. Esto nos lleva `(T-1)` comparaciones, una por cada thread.

   Conclusión : el coste total será de: $CT \in O(Q + T - 1) \in O(Q + T)$ operaciones.

2. **Dato no disponible en MM:**

   Situación de partida: en este escenario el thread menos especulativo quiere escribir un dato en memoria, por ello utiliza la operación del motor specstore_pointer().

   Primera operación: el thread número uno busca en la matriz principal (MM) el valor de la posición de memoria del dato a consultar. Tras llegar al último elemento, el `Q`, no se encuentra el valor, aun así, hemos necesitado `Q` comparaciones.

   Segunda operación: como el dato no está en memoria, se carga en la posición `Q + 1` de MM, lo que supone una operación más.

   Tercera operación: se copia el valor del dato que se quiere escribir, en la posición `Q + 1` del vector de versiones del thread menos especulativo.

   Conclusión : el coste total será de: $CT \in O(Q + 1 + 1) \in O(Q)$ operaciones.

7.6.3. Coste de la operación de consolidación parcial (a la matriz principal)

Situación de partida: en este escenario el thread más especulativo finaliza su ejecución y quiere proceder a consolidar sus datos en memoria, por lo que utiliza la operación del motor correspondiente.

Primera operación: el thread más especulativo debe esperar a que los `T-1` threads anteriores finalicen, esto produce un tiempo de espera, T_1. Imaginemos que durante la ejecución, el thread menos especulativo provoca que todos los demás invaliden su ejecución por squash. Cuando el thread no especulativo finaliza, hay que realizar `Q` operaciones, suponiendo que ha utilizado todos los punteros disponibles.

Segunda operación: cuando finaliza el thread no especulativo, pasamos a tener otro thread no especulativo, por lo que hay que esperar que este finalice, por ello sumamos un tiempo T_2. Cuando por fin finaliza, también provoca squash en todos los demás y se consolida, realizando `Q` operaciones más. Estas operaciones se repetirán hasta consolidar el thread inicialmente más especulativo, el que queríamos consolidar al principio.

Conclusión : el coste total será proporcional a la suma del tiempo de espera de las operaciones que requieran consolidación, esto es, $O(T_1 + Q + T_2 + Q + ... + T_T + Q) \in O(Q * T)$ y a los tiempos de espera de las operaciones que no requieran un commit, es decir, $(O(\sum_{i=0}^{T} T_i)$, esto es, $O(CT) \in O(Q * T + \sum_{i=0}^{T} T_i)$.

7.6.4. Coste de la operación de consolidación final (de la matriz principal a memoria)

el coste de esta operación depende del número de punteros utilizados, en nuestro caso `Q`, por tanto el coste total será de: $CT \in O(Q)$ escrituras en memoria.

Capítulo 8

Uso de matrices dinámicas de punteros

En este capítulo hablaremos de la solución que utiliza matrices dinámicas para albergar los datos, y el contenido de los punteros. Para ello aumentaremos la complejidad de la ventana deslizante existente.

8.1. Bases de la solución

En esta nueva versión, el motor contará con los siguientes elementos:

- **Ventana deslizante:** al igual que en la versión inicial del motor contaremos con una ventana deslizante, sin embargo, no tendrá el mismo aspecto que en la versión citada. Ahora la ventana estará formada por elementos de una estructura con cuatro componentes, a saber, el estado, un puntero a la matriz que almacena los datos, un indicador de la última fila con datos de la matriz citada y un indicador del tamaño de la matriz. Se puede ver una descripción gráfica en la figura 8.1, pero además a continuación describiremos cada uno de los componentes por separado:

  - **Estado:** como en la versión original del motor, aquí también es necesario conocer el estado de cada procesador (*running, free, done, pending_ squash* o *squash*), luego esta función es similar a la existente en la versión previa.
  - **Puntero a matriz de datos:** la principal novedad será la incorporación de un apuntador a una matriz de datos dinámica (descrita a continuación) que mantenga los valores necesarios para llevar a cabo el objetivo.
  - **Indicador de la última fila utilizada:** en realidad este indicador no es indispensable, ya que podría marcarse el final de la estructura con un "null". Sin embargo, esto obligaría a recorrer toda la estructura para llegar al fin de la misma. En su lugar, utilizamos el citado indicador.
  - **Indicador del total de filas de la matriz de datos:** se pretende hacer que el tamaño de las matrices sea dinámico, por tanto, cada thread debe conocer el tamaño máximo de su matriz actual.

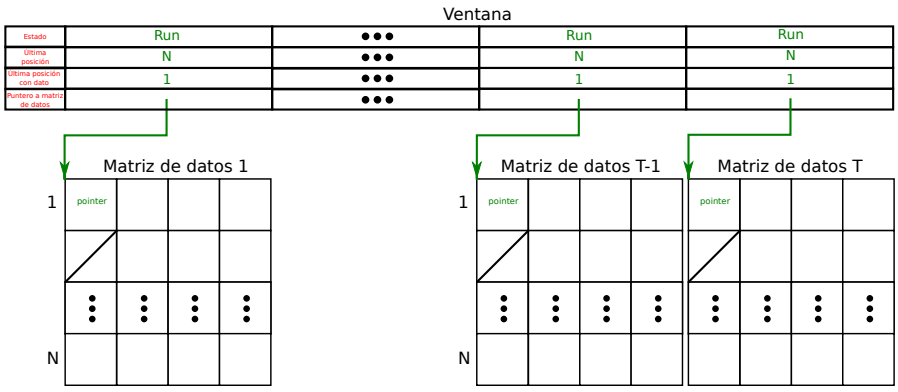

Figura 8.1: Solución 3: estructura de la nueva ventana deslizante.

- **Matrices dinámicas de cada thread:** la nueva versión del motor utilizará un nuevo tipo de estructuras, cada posición de la ventana poseerá una. Este tipo de matriz constará de `4 columnas` y de un `número de filas dinámico`. El tamaño de filas debe modificarse en tiempo de ejecución debido a que no conoceremos el número de datos a guardar en tiempo de compilación, y podemos necesitar más filas para guardar más valores. Podemos ver una descripción de la estructura en la figura 8.2.

  Las cuatro columnas que forman la matriz son las siguientes:

  1. **Puntero origen:** este elemento contendrá la dirección al dato original, es decir, la dirección del dato que leemos inicialmente, y sobre la que consolidaremos al final de la ejecución.
  2. **Tamaño de los datos:** este elemento contendrá el tamaño en bytes de los datos que apunta el puntero origen.
  3. **Puntero a la copia de los datos:** este elemento contendrá la dirección de los datos que almacenamos momentáneamente con la operación `specstore_pointer()`, es decir, nos sirve de almacén temporal, desechando su valor si el thread sufre una violación de dependencia, o almacenándolos en la dirección del puntero origen si el thread finaliza la ejecución con éxito.
  4. **Estado:** este elemento suplirá a la anterior matriz de acceso, es decir, mantendrá información sobre las operaciones a las que los datos han sido sometidas, a saber, `NotAcc, ExpLd, Updated, ElUp, RedAdd` y `RedMax`.

- **Apuntadores al thread no especulativo y al más especulativo:** estos elementos tienen el mismo significado que en la versión inicial del motor (ya vista en el capítulo 2), por tanto, se va a obviar su explicación.

En la figura 8.3 se pueden ver la nueva disposición del motor al completo, con las estructuras mencionadas.

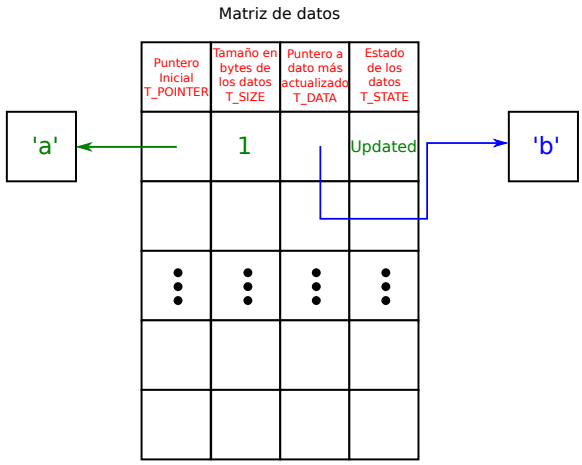

Figura 8.2: Solución 3: matriz de datos que utilizará el motor.

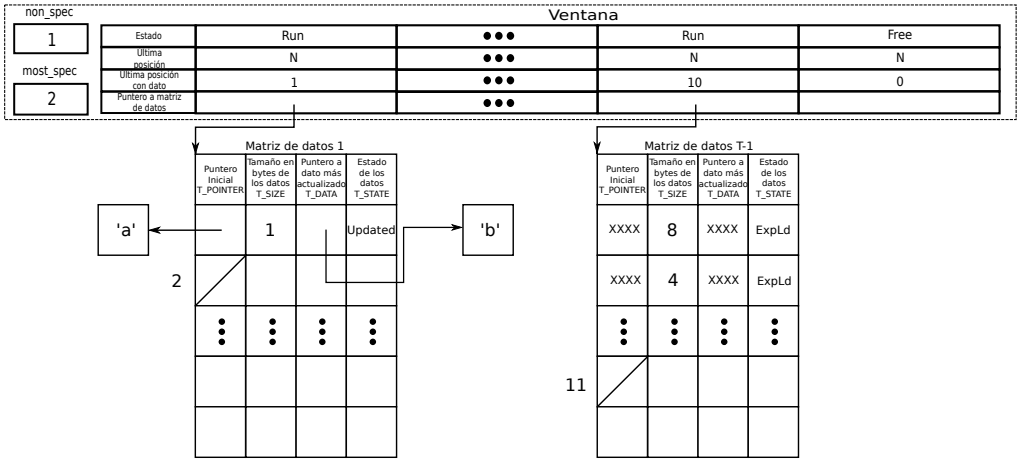

Figura 8.3: Solución 3: nueva disposición del motor de paralelización especulativa.

8.2. ¿Qué ocurre si la matriz de datos de un vector se completa?

El tamaño de las matrices se asigna de forma dinámica, así, si una matriz se llena de datos, el motor creará otra de mayor tamaño, asignará los elementos de la matriz inicial a los de la nueva, y liberará la matriz inicial. La ventaja de esta implementación es que el único thread que se verá afectado por estas operaciones es el que necesita más espacio de almacenamiento para su matriz, mientras que el resto seguirían con su tamaño inicial, sin sufrir efectos negativos en su tiempo de ejecución.

8.3. Modificación en las operaciones de lectura

Veamos el pseudocódigo de esta operación:

```
// Sea tid el thread en cuestion,
// pointer el valor del puntero buscado
// byteSize el tamaño en bytes de los datos referenciados por el puntero
// value la variable donde se cargan los datos

bool punteroUtilizado = FALSE;
unsigned int fila = 0;
int thread = tid;
// Primero comprueba si el thread en cuestión o
// alguno de los anteriores han utilizado el dato
while ( (punteroUtilizado == FALSE) && (thread>=wheel_ns) )
{
  if (wheel[thread].matrixPointer[T_POINTER][fila] == pointer)
    punteroUtilizado = TRUE;
  else
  {
    if (fila < wheel[thread].lastData)
      fila++;
    else
    {
      fila = 0;
      thread--;
    }
  }
}
// NOTA: en las variables thread y fila tendremos el valor del hilo que utilizó
// ese puntero y la posición que ocupa en su matriz (si ha sido utilizado)

// Compruebo si el thread actual ya había utilizado ese valor
if (tid == thread)
{
  if (wheel[tid].matrixPointer[T_STATE][fila] == UPDATE)
    wheel[tid].matrixPointer[T_STATE][fila] = ELUP;
  else
    wheel[tid].matrixPointer[T_STATE][fila] = EXPLD;
  // Compruebo el tamaño de los datos que contiene el puntero
  switch (wheel[tid].matrixPointer[T_SIZE][fila])
  {
```

```
    case 1:
      value = *(wheel[tid].matrixPointer[T_DATA][fila]);
    case 2:
      ...
    default:
      for (i=0;i<wheel[tid].matrixPointer[T_SIZE][fila];i++)
        value[i] = *(wheel[tid].matrixPointer[T_DATA][fila])[i];
  }
}
else
{
  int lastData = wheel[tid].lastData;
  // Compruebo que no se ha llenado la matriz de datos del thread en cuestión
  if (wheel[tid].tamMatrix == wheel[tid].lastData)
    reservarMasEspacioParaLaMatriz(wheel[tid]);
  wheel[tid].matrixPointer[T_STATE][lastData] = EXPLD;
  if (punteroUtilizado)
  {
    wheel[tid].matrixPointer[T_POINTER][lastData] =
                              wheel[thread].matrixPointer[T_POINTER][fila];
    wheel[tid].matrixPointer[T_SIZE][lastData] =
                              wheel[thread].matrixPointer[T_SIZE][fila];
    if (wheel[thread].matrixPointer[T_STATE][fila] == UPDATE)
      wheel[thread].matrixPointer[T_STATE][fila] = ELUP;

    // Reservo espacio para el dato, y guardo la dirección en la posición correspondiente
    wheel[tid].matrixPointer[T_DATA][lastData] =
                  (unsigned long int) malloc
                       (sizeof(wheel[thread].matrixPointer[T_SIZE][fila]));
    *wheel[tid].matrixPointer[T_DATA][lastData] =
                                *wheel[thread].matrixPointer[T_DATA][fila];
  }
  else
  {
    // Cargo el valor de memoria
    wheel[tid].matrixPointer[T_POINTER][lastData] = pointer;
    wheel[tid].matrixPointer[T_SIZE][lastData] = byteSize;
    // Reservo espacio para el dato, y guardo la dirección en la posición correspondiente
    wheel[tid].matrixPointer[T_DATA][lastData] =
                  (unsigned long int) malloc(sizeof(byteSize));
    *wheel[tid].matrixPointer[T_DATA][lastData] = *pointer;
  }
  switch (wheel[tid].matrixPointer[T_SIZE][lastData])
  {
    case 1:
      value = *(wheel[tid].matrixPointer[T_DATA][lastData]);
    case 2:
      ...
    default:
      for (i=0;i<wheel[tid].matrixPointer[T_SIZE][lastData];i++)
        value[i] = *(wheel[tid].matrixPointer[T_DATA][lastData])[i];
```

```
  }
  wheel[tid].lastData++;
}
```

La asimilación de este código puede ser muy farragosa en su entendimiento, por ello, procederemos a explicar un ejemplo, apoyándonos en imágenes. Supongamos que inicialmente tenemos en el motor los valores vistos en la figura 8.4(a). Imaginemos que en ese instante el thread dos ejecuta la siguiente instrucción:

```
A = *p;
```

El thread quiere obtener el valor de una posición de memoria. Supongamos que `p` vale 4 000, que `*p` vale 72 y que el dato apuntado por `p` es de cuatro bytes. ¿Cómo se llevaría a cabo la operación de lectura?

1. El thread en cuestión busca en su matriz de datos el valor de `p`: 4 000. Véase figura 8.4(b).
2. Como no se ha encontrado el valor de la posición de memoria (sólo contiene los valores 1 000, 2 500 y 2 000), significa que no ha utilizado anteriormente este puntero. Por tanto, mira en los threads predecesores en busca de si han utilizado ese puntero para obtener la versión más reciente del dato. Esto se describe en la figura 8.4(c).
3. Los threads anteriores no han utilizado ese dato, por tanto, se procede a cargar el valor directamente de memoria y se almacena en la primera posición vacía de la matriz de datos del thread dos. Esto se describe en la figura 8.4(d).

8.4. Modificación en las operaciones de escritura

Veamos el pseudocódigo de esta operación:

```
// Sea tid el thread en cuestion,
// pointer el valor del puntero buscado
// byteSize el tamaño en bytes de los datos referenciados por el puntero
// value el valor a almacenar

bool punteroUtilizado = FALSE;
unsigned int fila = 0;
int thread = tid;
// Primero comprueba si el thread en cuestión ha utilizado este valor
while ( (punteroUtilizado == FALSE) && (fila<=wheel[tid].lastData) )
{
  if (wheel[tid].matrixPointer[T_POINTER][fila] == pointer)
    punteroUtilizado = TRUE;
  else
    fila++;
}
// NOTA: En la variable fila tendremos la posición que ocupa el puntero
// en su matriz (si ha sido utilizado)
```

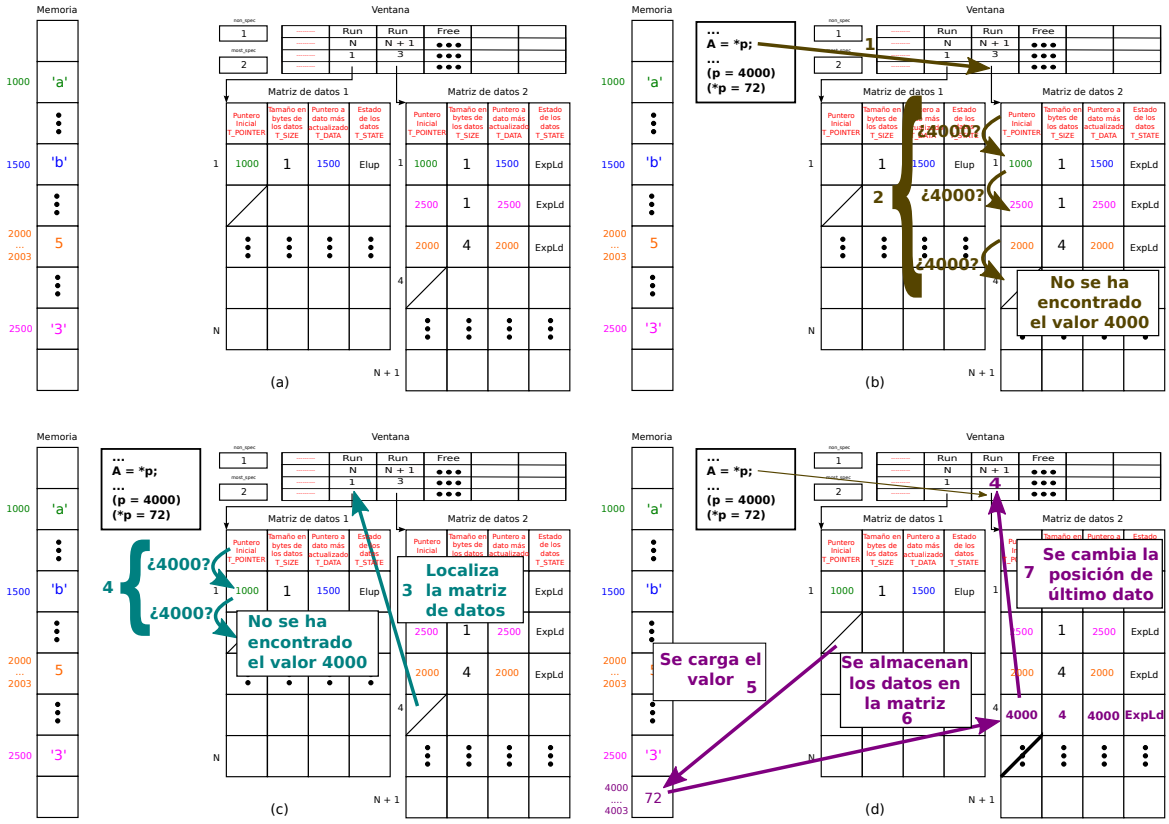

Figura 8.4: Solución 3: operación de lectura. (a) Valores iniciales de nuestro ejemplo. (b) El thread busca el valor del puntero en su matriz de datos. (c) El thread busca el valor del puntero en la matriz de datos de los threads predecesores. (d) Se carga el valor de memoria, y se almacena en la primera posición vacía de la matriz de datos del thread que lo solicitó. Hecho esto se aumenta en uno el indicador de la última posición.

```
// Compruebo si el thread actual ya había utilizado ese valor
if (punteroUtilizado)
{
  if (wheel[tid].matrixPointer[T_STATE][fila] == EXPLD)
    wheel[tid].matrixPointer[T_STATE][fila] = ELUP;
  else
    wheel[tid].matrixPointer[T_STATE][fila] = UPDATED;
  // Compruebo el tamaño de los datos que contiene el puntero
  switch (wheel[tid].matrixPointer[T_SIZE][fila])
  {
    case 1:
      *(wheel[tid].matrixPointer[T_DATA][fila]) = value;
    case 2:
      ...
    default:
      for (i=0;i<wheel[tid].matrixPointer[T_SIZE][fila];i++)
        *(wheel[tid].matrixPointer[T_DATA][fila])[i] = value[i];
  }
}
else
{
  int lastData = wheel[tid].lastData;
  // Compruebo que no se ha llenado la matriz de datos del thread en cuestión
  if (wheel[tid].tamMatrix == wheel[tid].lastData)
    reservarMasEspacioParaLaMatriz(wheel[tid]);
  wheel[tid].matrixPointer[T_STATE][lastData] = UPDATED;
  wheel[tid].matrixPointer[T_POINTER][lastData] = pointer;
  wheel[tid].matrixPointer[T_SIZE][lastData] = byteSize;
  // Reservo espacio para el dato, y guardo la dirección en la posición correspondiente
  wheel[tid].matrixPointer[T_DATA][lastData] =
                (unsigned long int) malloc(sizeof(byteSize));
  *wheel[tid].matrixPointer[T_DATA][lastData] =
                                  *wheel[thread].matrixPointer[T_DATA][fila];
  wheel[tid].lastData++;
}

// Ahora compruebo que ningún thread posterior haya utilizado el puntero,
// en cuyo caso, descartaremos su ejecución
thread = tid + 1;
fila = 0;
for (thread = tid + 1; thread<=wheel_ms; thread++)
{
  for (fila = 0; fila<wheel[thread].lastData; fila++)
  {
    if (wheel[thread].matrixPointer[T_POINTER][fila] == pointer)
    {
      // Existe el puntero
      wheel[thread].state = SQUASHED;
      break;
    }
  }
```

```
}
```

Entender el código expuesto directamente puede ser tedioso, por tanto, para describir la operación de escritura, recurriremos de nuevo a un ejemplo con imágenes. Supongamos que inicialmente tenemos en el motor los valores vistos en la figura 8.5(a). Imaginemos que en ese instante el thread número uno ejecuta la siguiente instrucción:

```
*p = datoP;
```

El thread quiere almacenar el valor de A de una posición de memoria. Supongamos que `p` vale 4 000, y que `datoP` es un dato de cuatro bytes. ¿Cómo se llevaría a cabo la operación de escritura?

1. El thread en cuestión busca en su matriz de datos el valor de `p`: 4 000. Véase figura 8.5(b).
2. Como no se ha encontrado el valor de la posición de memoria (sólo contiene los valores 1 000, 2 500 y 2 000), significa que no ha utilizado anteriormente este puntero. Por tanto, agrega el valor del dato a su matriz propia. Esto se describe en la figura 8.5(c).
3. Una vez hecho esto, se repasan las matrices de datos de los threads sucesores para comprobar que no han utilizado un dato incorrecto, en cuyo caso, se descarta la ejecución del thread. Esto se describe en la figura 8.5(d).

8.5. Modificación en las operaciones de consolidación

Veamos el pseudocódigo de esta operación:

```
// Sea tid el thread en cuestion,
// pointer el valor del puntero
// max será la primera posición de la matriz MM donde no haya puntero
// Q será el número de filas de la matriz MM
// valor es el valor a escribir

#pragma critical
if (wheel[tid].state != SQUASHED)
{
  if (tid == non_spec)
  {
    wheel[tid].state=DONE;
    for (i=non_spec; i<=most_spec; i++)
    {
      if (wheel[i].state == DONE && wheel[i+1].state != DONE)
      {
        last=i;
        break;
      }
    }
    for (j=non_spec; j<=last; j++)
    {
      for(k = 0; k < wheel[j].lastData; k++)
```

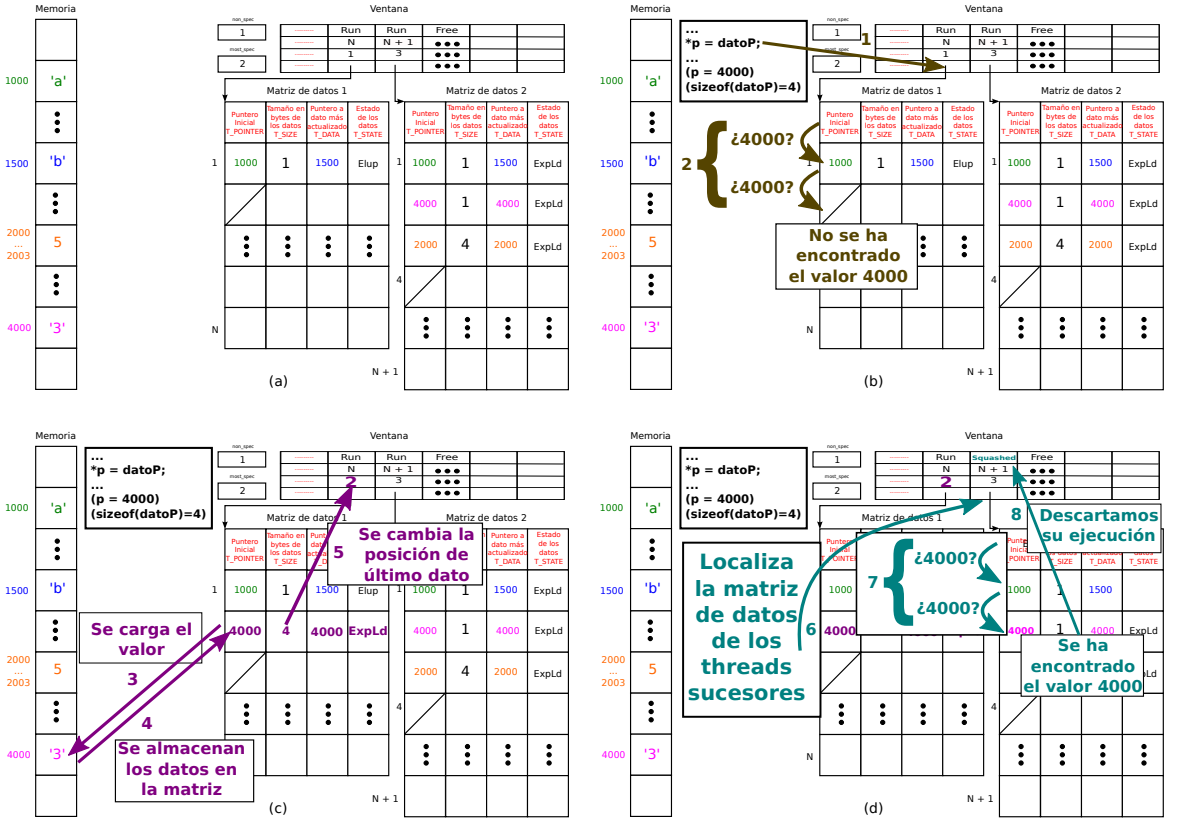

Figura 8.5: Solución 3: operación de escritura. (a) Valores iniciales de nuestro ejemplo. (b) El thread busca el valor del puntero en su matriz de datos. (c) El thread carga el valor del puntero en su matriz de datos. (d) Se comprueban en los threads sucesores si el valor había sido utilizado por alguno de ellos.

```
        *(wheel[j].matrixPointer[T_POINTER][k]) = *(wheel[j].matrizPointer[T_DATA][k]);
#pragma memory fence
      wheel[j].state = FREE;
    }
    non_spec = last+1;
  }
  else
    wheel[tid].state = DONE;
}
else
  do_squash();
#pragma end critical

while (wheel[most_spec+1].state != FREE) {}

#pragma critical
for (j=0; j<wheel[most_spec+1].lastData; j++)
  wheel[most_spec+1].matrixPointer[T_STATE][j] = NotAcc;
#pragma memory fence
wheel[most_spec+1].lastData = 0;
wheel[most_spec+1].state = RUN;
most_spec++;
#pragma end critical
```

Para el caso de la consolidación de resultados también veremos un ejemplo con imágenes. Supongamos que tenemos inicialmente los valores vistos en la figura 8.6(a) en las estructuras del motor especulativo. Supongamos que el thread dos finaliza el primero.

¿Cómo se llevaría a cabo la operación de consolidación?

1. La consolidación se realiza en orden, es decir, como el thread que finaliza su ejecución no es el no especulativo, debe esperar a que este termine. Por tanto, su única operación será el cambio de su estado a `DONE`. Podemos verlo en la figura 8.6(b).

2. Imaginemos que un tiempo después finaliza su ejecución el thread uno, es decir, el no especulativo. Este thread sí que puede consolidarse, como puede comprobarse en las flechas azules de la figura 8.6(c). Cuando ha consolidado sus datos, comprueba si hay algún thread sucesor esperando a realizar su commit (flecha roja número seis de la figura 8.6(c)), cosa que sucede, el thread dos esperaba a consolidar sus datos.

3. El thread dos ya puede consolidar sus datos, por tanto, realiza las operaciones pertinentes (flechas rosas y granates de la figura 8.6(d)). Hecho esto comprueba si algún thread sucesor espera a consolidarse (flecha roja número diez de la figura 8.6(d)), cosa que no sucede.

8.6. Optimización de la idea

Generalmente el puntero a los datos más actualizados será una dirección de memoria con un tamaño dependiente de la arquitectura, por ello, hemos considerado que si el tamaño de los datos apuntados (por el puntero referido) es menor que el tamaño de las direcciones de memoria (lo más habitual), almacene directamente el valor actualizado.

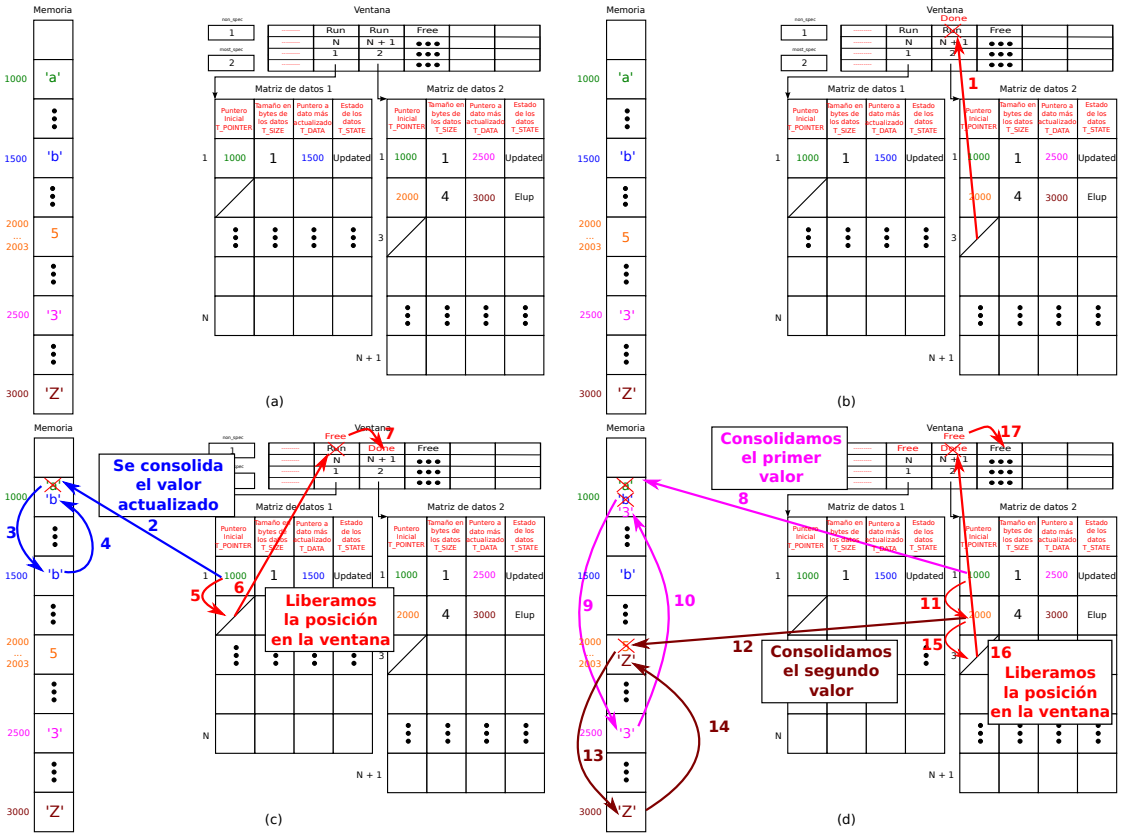

Figura 8.6: Solución 3: operación de consolidación. (a) Valores iniciales de nuestro ejemplo. (b) Inicialmente el thread dos finaliza, pero no puede consolidarse porque el thread no especulativo no ha finalizado su ejecución. (c) Consolidación de los valores del thread no especulativo. (d) Consolidación de los valores del thread dos tras el commit del thread no especulativo.

Esta modificación añade complejidad a la solución, sin embargo, nos ahorrará un acceso a memoria en la mayoría de ocasiones, es decir, reducirá el tiempo de ejecución.

El código que refleja esta idea será similar a lo siguiente:

```
...
// Siendo: T_SIZE el tamaño de los datos
//         baseType el tamaño de una dirección de memoria
if (T_SIZE <= sizeof(baseType) )
  value = T_DATA;
else
 value = *T_DATA;
...
```

8.7. Análisis de los costes

En esta sección se calculará el coste de las operaciones que ejecuta el motor, para ello intentaremos tomar escenarios con los peores casos posibles. Como datos de referencia del estudio, se tomarán `T` threads, `Q` punteros en todas las matrices, y `M` el número de filas en las matrices, tomando inicialmente en los supuestos `M=Q`.

8.7.1. Coste de la operación de lectura

Existen varias posibilidades a la hora de medir el coste de estas operaciones, a saber, si el dato ha sido, o no, leído anteriormente, es decir, si está en la matriz del thread en cuestión, o de alguno de los predecesores, o en ninguna matriz y debe ser cargado.

1. **Dato disponible en la matriz del thread inicial:**

   Situación de partida: en este escenario el thread más especulativo quiere leer un dato de memoria, por tanto, utiliza la función del motor specload_pointer().

   Primera operación: el thread número `T` busca en su matriz el valor de la posición de memoria del dato a consultar. Tras llegar al último elemento, el `Q`, encuentra el valor en este último elemento, lo cual ha supuesto `Q` comparaciones.

   Conclusión : el coste total será de: $CT \in O(Q)$ operaciones.

2. **Dato no disponible en la matriz del thread inicial, pero disponible en la matriz del thread no especulativo:**

   Situación de partida: en este escenario el thread más especulativo quiere leer un dato de memoria, por ello utiliza la operación del motor specload_pointer(). Supondremos que el thread no especulativo es el 1.

   Primera operación: el thread número `T` busca en su matriz el valor de la posición de memoria del dato a consultar. Tras llegar al último elemento, el `Q`, no encuentra el valor, lo cual ha supuesto `Q` comparaciones.

   Seguna operación: el thread número `T-1` busca en su matriz el valor de la posición de memoria del dato a consultar. Tras llegar al último elemento, el `Q`, no encuentra el valor, lo cual ha supuesto `Q` comparaciones más.

T-ésima operación: el thread número `1` busca en su matriz el valor de la posición de memoria del dato a consultar. Tras llegar al último elemento, el `Q`, encuentra el valor, lo cual ha supuesto `Q` comparaciones más.

Siguiente operación: el thread `T` carga el valor de ese dato a su matriz, sin embargo, no tiene espacio suficiente, por ello, debe reservar más memoria. La operación de reserva de memoria debe obtener una matriz con más memoria (`1` operación más), y asignar los datos de la anterior matriz a la nueva, en este caso `Q` asignaciones.

Operación final: se carga ese valor de la posición `Q` del thread `1` a la posición `Q + 1` del thread `T`.

Conclusión : el coste total será de: $CT \in O(Q * T + 1 + Q) \in O(Q * T)$ operaciones.

3. **Dato no disponible en la matriz del thread inicial, ni en la matriz de ningún otro thread:**

   Situación de partida: en este escenario el thread más especulativo quiere leer un dato de memoria, por ello utiliza la operación del motor specload_pointer(). Supondremos que el thread no especulativo es el 1.

   Primera operación: el thread número `T` busca en su matriz el valor de la posición de memoria del dato a consultar. Tras llegar al último elemento, el `Q`, no encuentra el valor, lo cual ha supuesto `Q` comparaciones.

   Seguna operación: el thread número `T-1` busca en su matriz el valor de la posición de memoria del dato a consultar. Tras llegar al último elemento, el `Q`, no encuentra el valor, lo cual ha supuesto `Q` comparaciones más.

   T-ésima operación: el thread número `1` busca en su matriz el valor de la posición de memoria del dato a consultar. Tras llegar al último elemento, el `Q`, no encuentra el valor, lo cual ha supuesto `Q` comparaciones más.

   Siguiente operación: el thread `T` carga el valor de ese dato a su matriz, sin embargo, no tiene espacio suficiente, por ello, debe reservar más memoria. La operación de reserva de memoria debe obtener una matriz con más memoria (`1` operación más), y asignar los datos de la anterior matriz a la nueva, en este caso `Q` asignaciones.

   Operación final: se carga ese valor de memoria a la posición `Q + 1` del thread `T`.

   Conclusión : el coste total será de: $CT \in O(Q * T + 1 + Q) \in O(Q * T)$ operaciones.

8.7.2. Coste de la operación de escritura

Hay dos posibilidades a la hora de medir el coste de estas operaciones, a saber, si el dato ha sido, o no, utilizado anteriormente por el thread que realiza la operación, es decir, si está en su matriz o no.

1. **Dato disponible en la matriz del thread:**

   Situación de partida: en este escenario el thread menos especulativo quiere escribir un dato en memoria, por ello utiliza la operación del motor specstore_pointer().

Primera operación: el thread número uno busca en su matriz el valor de la posición de memoria del dato a almacenar. Tras llegar al último elemento, el `Q`, encuentra el valor en este último elemento, lo cual ha supuesto `Q` comparaciones.

Segunda operación: en este punto debemos comprobar si el dato ha sido utilizado por algún thread posterior, para detectar posibles squashes. Esto nos lleva `Q*(T-1)` comparaciones, es decir, `Q` comparaciones por cada thread sucesor.

Conclusión : el coste total será de: $CT \in O(Q + Q * (T - 1)) \in O(Q * T)$ operaciones.

2. **Dato no disponible en la matriz del thread:**

   Situación de partida: en este escenario el thread menos especulativo quiere escribir un dato en memoria, por ello utiliza la operación del motor specstore_pointer().

   Primera operación: el thread número uno busca en su matriz el valor de la posición de memoria del dato a almacenar. Tras llegar al último elemento, el `Q`, no encuentra el valor, lo cual ha supuesto `Q` comparaciones.

   Segunda operación: el thread `1` intenta cargar el valor de ese dato a su matriz, sin embargo, no tiene espacio suficiente, por ello, debe reservar más memoria. La operación de reserva de memoria debe obtener una matriz con más memoria (`1` operación más), y asignar los datos de la anterior matriz a la nueva, en este caso `Q` asignaciones.

   Tercera operación: en este punto, una vez cargado el dato en la matriz del thread, debemos comprobar si el dato ha sido utilizado por algún thread posterior, para detectar posibles squashes. Esto nos lleva `Q*(T-1)` comparaciones, es decir, `Q` comparaciones por cada thread sucesor.

   Conclusión : el coste total será de: $CT \in O(Q + 1 + Q + Q * (T - 1)) \in O(Q * T)$ operaciones.

8.7.3. Coste de la operación de consolidación

Situación de partida: en este escenario el thread más especulativo finaliza su ejecución y quiere proceder a consolidar sus datos en memoria, por lo que utiliza la operación del motor correspondiente.

Primera operación: el thread más especulativo debe esperar a que los `T-1` threads anteriores finalicen, esto produce un tiempo de espera, T_1. Imaginemos que durante la ejecución, el thread menos especulativo provoca que todos los demás invaliden su ejecución por squash. Cuando el thread no especulativo finaliza, hay que realizar `Q` operaciones, suponiendo que ha utilizado todos los punteros disponibles.

Segunda operación: cuando finaliza el thread no especulativo, pasamos a tener otro thread no especulativo, por lo que hay que esperar que este finalice, por ello sumamos un tiempo T_2. Cuando por fin finaliza, también provoca squash en todos los demás y se consolida, realizando `Q` operaciones más. Estas operaciones se repetirán hasta consolidar el thread inicialmente más especulativo, el que queríamos consolidar al principio.

Conclusión : el coste total será proporcional a la suma del tiempo de espera de las operaciones que requieran consolidación, esto es, $O(T_1 + Q + T_2 + Q + ... + T_T + Q) \in O(Q * T)$ y a los tiempos de espera de las operaciones que no requieran un commit, es decir, $(O(\sum_{i=0}^{T} T_i)$, esto es, $O(CT) \in O(Q * T + \sum_{i=0}^{T} T_i)$.

Capítulo 9

Elección de la mejor solución entre las propuestas

Para obtener una versión nueva del motor de ejecución especulativa, necesitamos escoger la mejor de las soluciones posibles. Para ello a lo largo de este capítulo compararemos los costes de las soluciones vistas.

Para elegir una solución compararemos los costes que generan las soluciones citadas. La comparación se puede ver en el cuadro 9.1.

En la citada tabla podemos ver que, para las operaciones de lectura y escritura, el coste es menor para la solución basada en la matriz principal. Por otro lado, para las operaciones de consolidación, las soluciones uno y tres realizan `Q` operaciones menos.

En todo caso, podemos ver que los respectivos costes asintóticos no difieren excesivamente salvo en casos muy determinados. En consecuencia, la decisión de qué solución adoptar se basará en otras consideraciones, descritas en las secciones siguientes.

9.1. Inconvenientes de la solución 1

Contemplando la tabla 9.1 observamos que los costes son menores para esta solución que para las demás, sin embargo, este no es el único parámetro que debemos tener en cuenta. No debemos dejar en el olvido el propósito principal de el nuevo motor, la paralelización, por tanto, debemos recapacitar sobre la capacidad de paralelismo de las nuevas operaciones, es decir, el número de secciones críticas que aparecen.

Atendiendo a las características recién expuestas, y tras realizar un estudio sobre la solución basada en una matriz principal, se comprobó que los accesos concurrentes a la matriz citada serían un aspecto muy difícil de tratar, y deberían establecerse a través de secciones críticas para hacer que los threads puedan comunicarse sin errores. Como sabemos la aparición de una sección crítica ralentiza el tiempo de ejecución porque puede hacer que el paralelismo sea inexistente si todos los threads acceden a una al mismo tiempo. Por ello en una solución ideal el número de secciones críticas debería ser mínimo.

En la solución que estamos evaluando, las secciones críticas tienen una gran frecuencia de aparición, por que los accesos a la matriz principal son constantes. Por ello podemos afirmar que esta solución debe ser descartada.

Operación / Solución	Lectura	Escritura	Consolidación
Solución 1	$O(Q*T)$	$O(Q*T)$	$O(Q*T+\sum_{i=0}^{T} T_i)$
Solución 2	Dato disponible: $O(Q+T)$	Dato disponible: $O(Q+T)$	$O(Q*T+\sum_{i=0}^{T} T_i+Q)$
	Dato no disponible: $O(Q)$	Dato no disponible: $O(Q)$	
Solución 3	Dato disponible en thread actual: $O(Q*T)$	Dato disponible en thread actual: $O(Q*T)$	$O(Q*T+\sum_{i=0}^{T} T_i)$
	Dato disponible en thread no especulativo: $O(Q*T)$		
	Dato no disponible: $O(Q*T)$	Dato no disponible: $O(Q*T)$	

Cuadro 9.1: Comparación entre los costes producidos por la solución basada en matrices de versión de datos y matrices de punteros, la que se basa en una matriz global de punteros y varias matrices de versión de datos, y la que utiliza matrices de punteros dinámicas. Como referencia se utilizan `T` threads y `Q` punteros.

9.2. Inconvenientes de la solución 2

Para descartar esta solución atenderemos a otro criterio, la memoria. Podemos comprobar que la idea de mantener matrices dinámicas de punteros necesita de mucha menos memoria que la solución con una matriz estática. Imaginemos un caso extremo en que hay 4 threads, que el 1 utiliza 1 000 punteros, y los threads número 2, 3 y 4 utilizan 10 punteros cada uno. En la solución de las matrices estáticas necesitaremos que cada thread almacene 1 000 punteros, desperdiciando 990 posiciones en los threads 2, 3 y 4. Sin embargo, utilizando matrices dinámicas, el thread 1 tendrá una matriz de 1 000 posiciones, y los demás de 10 posiciones, optimizando la memoria reservada.

Además del citado, tenemos otra razón para creer que el uso de matrices dinámicas es mejor que las estáticas: no necesitamos saber en tiempo de compilación el número de punteros que va a utilizar un thread como máximo. Esto supone un gran ahorro en la implementación de la versión paralela especulativamente de una aplicación, penalizando mínimamente los tiempos de ejecución.

Tras descartar las otras soluciones, **concluimos utilizar la solución que implementa matrices dinámicas de punteros**.

Capítulo 10

Implementación del nuevo motor especulativo

Este capítulo describe cómo se llevó a cabo la implementación de la versión del motor con soporte para punteros. Para ello describiremos los problemas encontrados hasta lograr un funcionamiento correcto.

10.1. Introducción

Una vez realizado el desarrollo teórico, y comprobado que la idea era buena, comenzamos a desarrollar el código fuente de la nueva solución. El primer contratiempo con el que nos enfrentamos fue tomar la decisión de reutilizar el código existente o no.

10.2. ¿Deberíamos reutilizar el código de versiones previas?

Como ya se ha dicho esta cuestión fue la primera que debíamos tomar. Por tanto decidimos que para elaborar este nuevo motor, partiríamos de la base de no reutilizar apenas el código existente. Tomamos esta decisión porque las operaciones de las versiones previas apenas tendrían algo en común con las nuevas (salvo quizá el nombre) y las estructuras principales del motor cambiarían. Por los motivos citados, concluimos que reutilizar el motor ya conocido no serviría más que para confundirnos y partimos de cero en la implementación de la nueva arquitectura.

Cabe agregar en esta parte que se ha pensado en cambiar el nombre de todas las operaciones del motor para que no produzcan errores de compatibilidad con versiones previas, es decir, para que no se puedan utilizar en aplicaciones anteriores con tan sólo el cambio de ficheros del motor. Este cambio en la interfaz es requerido porque los argumentos serán diferentes, aunque no cambie la funcionalidad.

10.3. Implementación de las principales estructuras de datos

Como principales estructuras de datos queremos citar dos:

- **La ventana deslizante:** en las versiones iniciales la ventana deslizante se implementaba como un vector de enteros, y tan sólo almacenaba el estado de cada posición de la ventana. Sin embargo, en la versión actual es necesaria una funcionalidad mucho mayor.

  En este punto hemos decidido reutilizar la ventana existente en las versiones previas del motor, e implementar una estructura complementaria que soporte las demás funcionalidades. Con esto conseguimos desacoplar el estado de los demás datos, dando lugar a la reutilización de la ventana sin perder funcionalidades. El código `C` de las estructuras descritas es el siguiente:

```
// Definicion de la ventana deslizante
int wheel[wsize];

// Definicion de los datos de un slot de la ventana
typedef struct wheel_structure{
    threadCopy_structure *pointer;
    unsigned int lastData;
    unsigned int maxData;
}wheel_structure;

wheel_structure wheel_data[wsize];
```

  Cada posición de la nueva estructura está explicada en el capítulo anterior.

- **Las matrices de punteros:** para implementar estas matrices, se utilizará una estructura predefinida que tendrá `N` filas (dependiendo de la memoria que queramos reservar) y 4 columnas. Las columnas se definirán con constantes cuya definición en `C` es:

```
#define T_POINTER 0
#define T_SIZE 1
#define T_DATA 2
#define T_STATE 3
```

  Además para facilitar la legibilidad hemos decidido definir una nueva estructura, a saber, `threadCopy_structure` cuyo código es:

```
typedef struct threadCopy_structure {
    baseType matriz[4][MAX_POINTER];
} threadCopy_structure;
```

  Como se puede ver el tipo base de la matriz se define también como un tipo predefinido, `baseType`. Hemos definido este tipo para garantizar que el software funcione tanto en arquitecturas de 32, como de 64 bits. Si la arquitectura es de 32 bits, entonces los punteros tendrán un tamaño de 4 bytes, y por lo tanto los elementos de la "matriz de datos" deberán definirse como "unsigned long int" para que también tengan 4 bytes. Por otro lado, si la arquitectura es de 64 bits, los punteros tendrán ocho bytes, y habrá que definir los elementos de la "matriz de datos" como "unsigned long long int". Para realizar esta tarea hemos incluido una directiva de "compilación condicional" que defina el tipo base de los elementos de la "matriz de datos". A continuación se muestra la definición en `C`:

```
// Definicion de un tipo base para que funcione tanto en arquitecturas
// de 32 bits, como de 64 bits
#ifdef __LP64__
  typedef unsigned long long int baseType;
#else
  typedef unsigned long int baseType;
#endif
```

10.4. Implementación de la nueva operación de lectura especulativa

La nueva operación de lectura especulativa será radicalmente diferente a la existente anteriormente, en concreto ahora se denominará `specload_pointer()`. Esta nueva función se describe del siguiente modo:

specload_pointer(address, byteSize, current, value)

Los argumentos tienen el siguiente significado:

1. **address:** la dirección del puntero que se desea cargar, pasada como un puntero de tamaño uno.
2. **byteSize:** el tamaño que ocupa en memoria lo apuntado por el puntero pasado por el argumento anterior.
3. **current:** el thread actual que lleva a cabo la operación. Este argumento ya existía en la función `specload()` de las versiones previas del motor.
4. **value:** la dirección del puntero al elemento donde deseamos cargar el dato apuntado por `address`. Al igual que el ya mencionado, este argumento se pasa como un puntero de tamaño uno.

No entraremos a exponer aquí un ejemplo de llamada a esta función dado que ya ha sido descrito en la sección 8.3.

10.5. Implementación de la nueva operación de escritura especulativa

La operación de escritura especulativa al igual que ocurría con la de lectura, será distinta a la de versiones previas. También cambia su nombre, ahora se denominará `specstore_pointer()`. Esta nueva función se describe del siguiente modo:

specstore_pointer(address, byteSize, current, value)

Los argumentos tienen el siguiente significado:

1. **address:** la dirección del puntero donde se desea almacenar un valor. Se debe pasar como un puntero de tamaño uno.
2. **byteSize:** el tamaño que ocupa en memoria lo apuntado por el puntero pasado por el argumento anterior.
3. **current:** el thread actual que lleva a cabo la operación. Este argumento ya existía en la función `specstore()` de las versiones previas del motor.

4. **value:** la dirección del puntero al elemento del que queremos almacenar el valor en `address`. Al igual que el ya mencionado, este argumento se pasa como un puntero de tamaño uno.

En este caso tampoco entraremos a proporcionar un ejemplo de llamada a esta función porque ha sido descrito en la sección 8.4.

10.6. Implementación de la nueva operación de consolidación

En el caso de esta operación, conceptualmente apenas existen diferencias con la operación de consolidación de las versiones previas, sin embargo al igual que con las funciones ya citadas en este capítulo, también modifica su nombre, ahora denominado `threadend_pointer()`, y los argumentos de llamada. Esta nueva función se describe del siguiente modo:

threadend_pointer(current, nonSpeculative)

Los argumentos tienen el siguiente significado:

1. **current:** el thread actual que lleva a cabo la operación. Este argumento ya existía en la función `threadend()` de las versiones previas del motor.

2. **nonSpeculative:** sirve como indicador para saber si el thread que realiza la llamada a la función es el no especulativo antes de comenzar el trabajo.

Si el lector desea ver una demostración de cómo se llevaría a cabo la ejecución, podemos acudir a la sección 8.5.

10.7. Operaciones de reducción

Debido al tiempo disponible no se han podido implementar estas operaciones, cuyo diseño quedará pendiente como trabajo futuro.

Capítulo 11

Pruebas

A lo largo de este capítulo se entra en los procedimientos que se llevaron a cabo para realizar las pruebas del software desarrollado. Abordaremos pruebas con diferentes tamaños de bloque, y con diferente número de threads.

11.1. Introducción

Como indica Pressman [15], *las pruebas del software son un elemento crítico para la ganancia de calidad del software y representa una revisión final de las especificaciones, del diseño y de la codificación.*

Sabemos que las pruebas software son necesarias para evitar errores futuros en las aplicaciones desarrolladas, en este caso recurriremos a ellas para verificar la validez y robustez de la aplicación desarrollada.

Para elaborar las pruebas hemos adaptado la aplicación de ejemplo de la versión original, el fichero `ejemplo-vBRA09.c` descrito en el capítulo 2, obteniendo el fichero `ejemplo-v10.c` descrito a continuación:

```
#include <stdio.h>
#include <stdlib.h>
#include "miejemplovariables.h"

//*****************************************************************
// speccode: OpenMP header file included
#include <omp.h>
//*****************************************************************
//*****************************************************************
// speccode: Main header file with all common variables
#include "specEngine.h"
//*****************************************************************

int main() {

    // Variables locales
    // P indica la iteraccion en la que estamos, Q indica el indice del vector
```

```
int P, Q, aux, i;
FILE *fichero;
int sumatorio1 = 0;
int sumatorio2 = 0;
int vector2[MAX];
int aux_fscanf;

specinit();

// Apertura y lectura del fichero

if ((fichero = fopen("rand1000000.in", "r")) == NULL) {
    printf("Error en apertura del fichero para lectura \n ");
    exit(0);
} else {
    aux_fscanf = fscanf(fichero, "%d", &aux);
    for (i = 0; i < MAX; i++) {
        vector[i] = aux;
        vector2[i] = aux;
        aux_fscanf = fscanf(fichero, "%d", &aux);
    }
    fclose(fichero);
}

// miejemplocode: debug
//  for ( i = 0; i < MAX; i++)
//    printf("%d\n", vector[i]);
// miejemplocode: end debug

// El bucle no es paralelizable porque los indices
// de los elementos del vector donde escribimos
// dependen de los valores de la entrada del fichoer
// ESTO ES LO QUE SE BUSCA
// EMPEZAMOS

//*************************************************************************
// speccode: OMP threading directive
omp_set_num_threads(threads);
// speccode: initialize speculation structures
specstart(NITER);
// speccode: variable upper limit inicialization
//upper_limit = NITER;
//*************************************************************************
//*************************************************************************
// speccode: Start profiling
// collector_resume();
//*************************************************************************
// speccode: Speculative loop
int iteration, ini, nIterations;
int value = 0;
int linear = 0;
```

```
#pragma omp parallel default(none)                                    \
               private(Q,P,aux,linear, iteration, ini,nIterations, value, \
                       current, tid, retflag, stdout, nonSpeculative)    \
              shared(vector, wheel_ns, wheel_ms,    \
                     wheel, upper_limit, varblock, endLoop, endReturn)
    {

#pragma omp for \
   schedule(static)
        //for ( P = 1 ; P <= NITER ; P++ )
        //initLoopSpecEngine(vector,P,0,1)
        /**
         * @author Alvaro Estebanez Lopez
         * @arg iteration index of the iteration is being excuted inside the
         *      speculative loop
         * @arg ini initial index of the loop
         * @arg IDLOOP loop identifier
         * Assigning the first block each thread
         * According to flag exceptions checkPoint is enabled or disabled
         **/
        //IDLOOP = 1;
        //#define initLoopSpecEngine(iteration,ini,IDLOOP)

        //iteration = P;
        for (tid = 0; tid <= threads - 1; tid++) {

            ini = 0;
            if (getFirstBlock(&current, &nonSpeculative) == 0) {
                goto labelEndLoop;
            }
            iteration = varblock[0][current] + ini;
            varblock[2][current] = varblock[2][current] + 1;
labelStartIteration:
            //  Codigo del bucle

            Q = iteration % (MAX) + 1;

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

```
            Q = (4 * aux) % (MAX) + 1;

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

            //endLoopSpecEngine(vector,P,NITER,0,1);
            //endLoopSpecEngine(i, cur_upper_limit, 0, 1);
            //iteration = P;
            nIterations = NITER;
            ini = 0;
            /**
             * @author Alvaro Estebanez Lopez
             * @arg iteration index of the iteration is being excuted
             *      inside the speculative loop
             * @arg nIterations total number of iterations in the loop
             * @arg ini initial index of the loop
             * @arg IDLOOP loop identifier
             * Returning point after ending an iteration
             * A new block of iterations will be assigned providing that
             * there are still work to execute
             **/
            //#define endLoopSpecEngine(iteration, nIterations, ini, IDLOOP)
            /************************************************************* \
            // speccode: end of thread and commit \
            //           First check if block of iterations is done **********/ \
if (endLoop || endReturn) {
                freeWheel(current);
                goto labelEndLoop;
            }
labelEndIteration:
            if ( (iteration != varblock[1][current] + ini)
                && (iteration < nIterations - 1) ) {
                iteration = iteration + 1;
                goto labelStartIteration;
            }
            /* Thread done, perform the commit */
labelSquash:
            retflag = threadend_pointer(&current, &nonSpeculative);
            /* End if job done */
            if (retflag == JOBDONE) goto labelEndLoop;
            if (retflag == JOBTODO) {
                /*  Set loop variable */
                iteration = varblock[0][current] + ini;
```

```
                varblock[2][current] = varblock[2][current] + 1;
                goto labelStartIteration;
            }
labelEndLoop:
            ; /*freeWheel(current);*/
        } /*End for initLoop()*/

    } // #pragma omp parallel

    // Obtenemos una version secuencial para comprobar fallos

    for (P = 0; P <= NITER; P++) {


        Q = P % (MAX) + 1;
        aux = vector2[Q - 1];

        Q = (4 * aux) % (MAX) + 1;
        vector2[Q - 1] = aux;
    }

    // Escritura del resultado del vector
    sumatorio1=0;
    printf(" Resultado del vector1\n");
    for (i = 0; i < MAX; i++)
        sumatorio1 = sumatorio1 + vector[i];
    printf("suma vector1=%d\n", sumatorio1);

    sumatorio2=0;
    printf(" Resultado del vector2\n");
    for (i = 0; i < MAX; i++)
        sumatorio2 = sumatorio2 + vector2[i];
    printf("suma vector2=%d\n", sumatorio2);

    if (sumatorio1 != sumatorio2) {
        for (i = 0; i < MAX; i++) {
            if (vector[i] != vector2[i])
                printf("vector[%d]=%d, vector2[%d]=%d,\n", i, vector[i], i, vector2[i]);
        }
    }
    return 0;
}
```

Como se puede apreciar en el código mostrado, podemos ver que en la misma aplicación realizamos el sumatorio especulativamente y secuencialmente, para después comparar los resultados de ambos sumatorios, y ver en qué elementos difieren los resultados.

11.2. Pruebas con diferente número de threads

En el cuadro 11.1 mostramos los resultados obtenidos al realizar las pruebas con un número distinto de procesadores. Para realizar las pruebas utilizamos un tamaño de bloque de 1000

Número de threads	Salida esperada	Número de errores	Resultado
1	vector=1642098	0/10	Negativo
2	vector=1642098	0/10	Negativo
3	vector=1642098	0/10	Negativo
4	vector=1642098	0/10	Negativo
5	vector=1642098	0/10	Negativo
6	vector=1642098	0/10	Negativo
7	vector=1642098	0/10	Negativo
8	vector=1642098	0/10	Negativo
9	vector=1642098	0/10	Negativo
10	vector=1642098	0/10	Negativo
11	vector=1642098	0/10	Negativo
12	vector=1642098	0/10	Negativo
13	vector=1642098	0/10	Negativo
14	vector=1642098	0/10	Negativo
15	vector=1642098	0/10	Negativo
16	vector=1642098	0/10	Negativo

Cuadro 11.1: Pruebas de ejecución utilizando un número diferente de threads.

iteraciones y ejecutamos la aplicación en diez ocasiones.

11.2.1. Resultados de las pruebas

De lo visto en el cuadro 11.1 podemos afirmar que el motor se comporta bien con cualquier número de threads.

11.3. Pruebas con diferentes tamaños de bloque

A lo largo de esta sección remitiremos una batería de pruebas similar a la descrita con anterioridad, sin embargo, ahora no trascenderá el número de procesadores con que ejecutaremos la aplicación, sino el tamaño del bloque de iteraciones que asignaremos a cada procesador. Para realizar estar pruebas utilizamos cuatro procesadores. Podemos ver los resultados en el cuadro 11.2.

11.3.1. Resultados de las pruebas

Observando los resultados del cuadro 11.2 vemos que la aplicación falla más cuando los bloques son pequeños. En este caso las pruebas nos fueron muy útiles ya que nos permitió detectar fallos escondidos en la implementación del código del motor.

11.3.2. Pruebas tras la primera revisión del código del motor

Tras observar los resultados positivos de las pruebas decidimos revisar la implementación de la nueva arquitectura del motor en busca de algún error conceptual. Encontramos varias

Tamaño de bloque	Salida esperada	Número de errores	Resultado
5	vector=1642098	7/10	Positivo
10	vector=1642098	5/10	Positivo
50	vector=1642098	2/10	Positivo
100	vector=1642098	1/10	Positivo
200	vector=1642098	0/10	Negativo
500	vector=1642098	0/10	Negativo
1000	vector=1642098	0/10	Negativo
2000	vector=1642098	0/10	Negativo
3000	vector=1642098	0/10	Negativo
4000	vector=1642098	0/10	Negativo
5000	vector=1642098	0/10	Negativo
6000	vector=1642098	0/10	Negativo
8000	vector=1642098	0/10	Negativo
10 000	vector=1642098	0/10	Negativo

Cuadro 11.2: Pruebas de ejecución utilizando un tamaño distinto de bloques de iteraciones.

Tamaño de bloque	Salida esperada	Número de errores	Resultado
5	vector=1642098	10/20	Positivo
10	vector=1642098	5/20	Positivo
50	vector=1642098	0/20	Negativo
100	vector=1642098	0/20	Negativo
200	vector=1642098	0/20	Negativo
500	vector=1642098	0/20	Negativo

Cuadro 11.3: Pruebas de ejecución utilizando un tamaño distinto de bloques de iteraciones tras solventar algunos errores en el código.

cosas que se podían mejorar, y tras solventar dichas erratas nos propusimos a realizar otra batería de pruebas que podemos observar en el cuadro 11.3. Para estas pruebas realizamos veinte ejecuciones con cada parámetro y nos centramos en los casos límite, es decir, los que provocaban errores. El número de procesadores utilizados seguirán siendo cuatro.

11.3.3. Resultados de las pruebas

Al igual que lo acontecido en las pruebas anteriores, el motor seguía sin ser correcto al 100 %, y seguía fallando en los casos más extremos (véase cuadro 11.3).

11.3.4. Pruebas tras la segunda revisión del código del motor

De nuevo teníamos resultados positivos en las pruebas, por tanto, al igual que antes decidimos realizar una búsqueda más exhaustiva de errores en el código del motor. Tras varias revisiones logramos encontrar sentencias que podían causar los fallos experimentados. Podemos ver los resultados en el cuadro 11.4. Para estas pruebas realizamos treinta ejecuciones

Tamaño de bloque	Salida esperada	Número de errores	Resultado
5	vector=1642098	0/30	Negativo
7	vector=1642098	0/30	Negativo
10	vector=1642098	0/30	Negativo

Cuadro 11.4: Pruebas de ejecución utilizando un tamaño distinto de bloques de iteraciones tras solventar más errores en el código.

con cada parámetro y nos centramos de nuevo en los casos límite, es decir, los que provocaban errores en la primera revisión del código. El número de procesadores que utilizamos continúa siendo cuatro.

11.3.5. Resultados de las pruebas

Los resultados obtenidos en estas pruebas (véase el cuadro 11.4) nos permiten afirmar que para este caso de ejemplo, nuestro motor se comporta correctamente.

Tras obtener unos resultados satisfactorios, decidimos dar por concluida la fase de pruebas.

Capítulo 12

Descripción y adaptación de benchmarks

A continuación expondremos cómo hemos abordado la paralelización especulativa de ciertos bechmarks, con la nueva arquitectura propuesta. Además aportaremos una breve descripción de las aplicaciones prácticas de citados benchmarks.

12.1. Algoritmos incrementales aleatorizados

Los algoritmos incrementales aleatorizados han sido ampliamente estudiados en áreas como la Geometría Computacional y la Optimización. Su uso ha permitido desarrollar algoritmos simples, fáciles de implementar y eficientes para una gran variedad de problemas, entre los que podemos citar la intersección de segmentos rectilíneos, los diagramas de Voronoi, las triangulaciones de polígonos simples, la programación lineal y muchos otros.

En su formulación más general, la entrada de un algoritmo incremental aleatorizado es un conjunto de elementos (no necesariamente puntos), para los que debe calcularse una cierta salida. El algoritmo procede de manera incremental, generalmente con un bucle que itera sobre todos los elementos, y su principal característica es que los elementos se van insertando según un orden aleatorio. Para que dos iteraciones puedan calcularse simultáneamente en distintos procesadores, no deben existir dependencias entre los resultados calculados en la primera y los datos utilizados en la segunda. Independientemente del problema que resuelvan, estos algoritmos presentan un patrón similar de dependencias entre iteraciones del bucle (como veremos en detalle en la próxima sección). De manera informal podemos decir que, al principio de la ejecución, muchos elementos que se insertan cambian la solución que se está calculando, de la cual depende la inserción de los siguientes elementos. Sin embargo, a medida que la ejecución avanza, cada vez aparecen menos dependencias entre iteraciones, es decir, menos elementos modifican la solución. Por un lado esto hace que la complejidad esperada de los algoritmos incrementales aleatorizados sea notablemente más baja que la complejidad en el caso peor. Por otra parte, esta distribución de dependencias hace de la paralelización especulativa la mejor técnica para su ejecución en paralelo.

Resumimos a continuación los problemas de este tipo que abordaremos para la obtención de resultados experimentales:

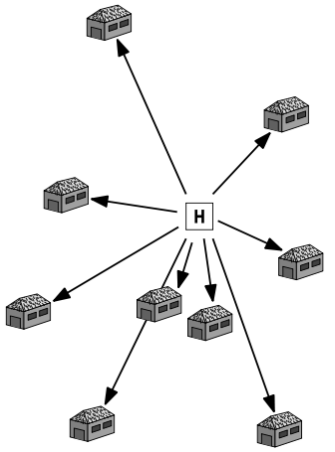

Figura 12.1: Hospital que presta servicio a varias comunidades.

- El algoritmo de Welzl para calcular el Menor Círculo Contenedor de una nube de puntos en el plano.
- El algoritmo de Clarkson et al. para el cálculo de la Envolvente Convexa.

12.2. Algoritmo para el cálculo del menor círculo contenedor

12.2.1. Introducción al problema

El problema del *menor círculo contenedor* [9] es del tipo *spanning*, esto es, de encontrar una figura que contenga a todos los puntos. Este problema es muy común cuando se quiere planificar la localización de recursos compartidos de tal manera que un objeto debe estar lo más cerca de todos los demás. Para entenderlo mejor, pensemos en un entorno fuera de la informática, por ejemplo, un hospital que presta servicio a varias comunidades, véase figura 12.1. Si pensamos en las comunidades como puntos del plano, el centro del menor círculo contenedor que englobe a todas ellas será un buen sitio donde colocar el hospital, minimizando la distancia entre el hospital y las comunidades más alejadas.

El algoritmo más simple considera que todos los círculos estarán definidos por dos o tres, de los `n` puntos del conjunto, véase figura 12.2. Cada círculo generado es $O(n^3)$, y cada comprobación es $O(n)$, con un tiempo total de ejecución de $O(n^4)$.

En este punto cabe señalar que exactamente no es exactamente el algoritmo de Welzl con el que nosotros trabajaremos, sino que para simplificar el problema hemos utilizado un algoritmo iterativo basado en el recursivo de Welzl. El algoritmo iterativo, sacado de [9], es el siguiente:

```
/*
 * Calcula el menor círculo contenedor de un conjunto de puntos de manera iterativa.
 * Inicialmente:
 *  P = conjunto con puntos en el plano.
 *  n = número de puntos que contiene P
```

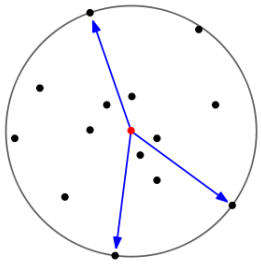

Figura 12.2: Menor círculo contenedor definido por tres puntos.

```
 */

función iMCC(P)
{
    D := Conjunto vacío;
    Para i:=1 hasta n
        p1 := punto i de P
        si ( p1 no está dentro del círculo definido por D ) entonces
            D := D U {p1};
            Para j:=1 hasta (i-1)
                p2 := punto j de P;
                si ( p2 no está dentro del círculo definido por D ) entonces
                    D := D U {p1,p2};
                    Para k:=1 hasta (j-1)
                        p3 := punto k de P;
                        si ( p3 no está dentro del círculo definido por D ) entonces
                             D := D U {p1,p2,p3};
                        fin_si
                    fin_para
                fin_si
            fin_para
        fin_si
    fin_para
    devolver D;
}
```

Dada una nube de puntos $P \subseteq \mathbb{R}^2$, denotamos por $D(S)$ su menor círculo contenedor, definido por los tres puntos (a lo sumo) por los que pasa. Llamamos R_i a los i primeros puntos de la nube, y supongamos calculado $D(R_{i-1})$. Al añadir el $i-$ésimo punto p_i pueden pasar dos cosas:

- Si p_i está dentro de $D(R_{i-1})$, entonces $D(R_i) = D(R_{i-1})$.
- Si p_i está fuera de $D(R_{i-1})$, se debe descartar $D(R_{i-1})$. A cambio, se obtiene que p_i está en la frontera. Cada candidato $p_j \in R_{i-1}$ se selecciona de manera similar a lo hecho para p_i. Análogamente, elegido un p_j puede ser necesario buscar en R_{j-1} un tercer punto p_k.

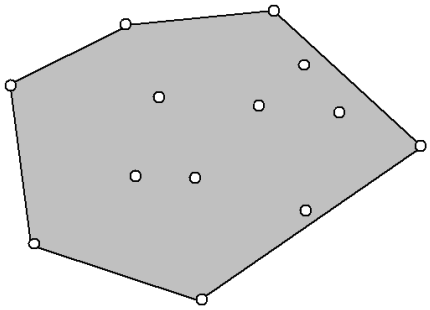

Figura 12.3: Cierre convexo de una nube de puntos.

12.2.2. Implementación

Para desarrollar este benchmark se ha recurrido a la versión más reciente disponible, es decir, una versión tomada de [5] que ya estaba paralelizada especulativamente mediante el motor original. Cabe mencionar que, como se puede haber percibido en la subsección anterior, el algoritmo consta de tres bucles anidados, y los tres son susceptibles de ser paralelizables. En el trabajo mencionado hay versiones paralelas de los tres, sin embargo, para nuestras pruebas nos limitaremos a la versión que paraleliza el bucle interior de la aplicación.

Las modificaciones realizadas en el código existente han sido tan sólo las adaptaciones de las operaciones pertinentes (lecturas y escrituras especulativas, y consolidación de datos) y no se han encontrado problemas reseñables.

12.3. Algoritmo para el cálculo del envolvente convexo

12.3.1. Introducción al problema

La envolvente convexa, también denominada cierre convexo o *convex hull*, es uno de los más fundamentales constructores geométricos. Una idea intuitiva del significado del cierre convexo es el contenido de la figura que formaría una banda elástica que rodeara a una nube de puntos una vez que la soltáramos. Podemos ver un ejemplo en la figura 12.3.

El problema de computar un *convex hull* no sólo está centrado en aplicaciones prácticas, sino también es un vehículo para la solución de un número de cuestiones aparentemente sin relación con él, que surgen en la Geometría Computacional. La computación del cierre convexo de una nube finita de puntos, especialmente en el plano, ha sido exhaustivamente estudiada y tiene aplicaciones, como por ejemplo, en el procesado de imágenes y en localización. Desafortunadamente, no es posible construir la definición intuitiva de cierre convexo citada anteriormente de forma natural, por lo que hay que identificar las nociones apropiadas que nos conduzcan a un algoritmo.

Graham en 1972 dio con el primer algoritmo que resolvía el problema del cierre convexo con un conjunto de puntos en un orden $O(n \log n)$ y un espacio de orden $O(n)$. Este algoritmo lo que hacía es coger los tres primeros puntos y calcular el baricentro. Ordenaba los puntos del polígono angularmente respecto de este baricentro y por ultimo recorría la lista ordenada de puntos eliminando los vértices que no son convexos. Este algoritmo se ha ido estudiando desde entonces y se han conseguido diferentes versiones en dos y hasta en tres dimensiones.

En este punto cabe señalar que nosotros utilizaremos el algoritmo de Clarkson *et al.* descrito a continuación: dada una nube S de puntos en el plano, se denota por $CH(S)$ la lista ordenada de vértices de su envolvente convexa. El algoritmo de Clarkson *et al.* consiste en un bucle principal que itera sobre cada uno de los puntos de la nube en un orden aleatorio: dada una permutación aleatoria R de los puntos de S, llamamos R_i al conjunto de los i primeros puntos de la permutación. Supongamos que el bucle ha avanzado hasta el i-ésimo punto p_i. Si p_i está dentro de $CH(R_i - 1)$ (ya calculada), entonces $CH(R_i) = CH(R_i - 1)$ y la iteración termina. En caso contrario (ver la Figura 1), $p_i \in CH(R_i)$, por lo que se deben calcular las dos tangentes a $CH(R_i - 1)$ desde p_i, eliminar de la lista $CH(R_i - 1)$ los vértices entre los puntos de tangencia (si los hay) y añadir p_i a la lista en su sustitución.

Para el estudio de la complejidad del algoritmo, es necesario tener en cuenta que se utiliza una estructura de datos que almacena información sobre la solución parcial a medida que se va construyendo. Esta estructura permite realizar la localización de un punto en tiempo esperado $O(\log n)$. Por otro lado, es fácil ver que cada punto puede dar lugar como máximo a dos aristas que se pueden borrar una única vez, con lo que la complejidad esperada del algoritmo de Clarkson *et al.* es $O(n \log n)$.

12.3.2. Implementación

Al igual que en la aplicación anterior, se disponía de una versión que paralelizaba la aplicación actual mediante el motor especulativo original.

En este caso las modificaciones también se han limitado a cambios simples en la nomenclatura de las funciones (lecturas y escrituras especulativas, y consolidación de datos) y tampoco se han encontrado dificultades a mencionar.

Capítulo 13

Resultados experimentales

A lo largo de este capítulo trataremos de comparar la versión elaborada del motor frente a la versión original del mismo, con vistas a poder establecer unas conclusiones sobre los tiempos de ejecución de cada arquitectura. Para realizar este cometido se utilizarán los benchmarks desarrollados y comentados en el capítulo anterior, así como la aplicación sintética utilizada para las pruebas del motor.

13.1. Introducción

A continuación presentaremos un conjunto de tablas mostrando los resultados obtenidos para la ejecución de cada aplicación, en concreto, mostraremos el tiempo de cada ejecución con un número diferente de procesadores asignados.

13.2. Entorno de ejecución

Lo primero que hay que señalar es que se ha utilizado el compilador `GCC v4.4.3` en `Ubuntu Linux`.

Las ejecuciones y pruebas de la aplicación se han desarrollado en un servidor propio del grupo de investigación: *Nodoyuna*. Se procederá a su descripción para poder interpretar correctamente los resultados de las pruebas:

- **Nodoyuna**: esta máquina es un servidor del tipo `Intel Dual Core` equipado con un procesador `quad-core Intel Core2 Q6600` a 2.40GHz, 4096 KB de memoria caché y 2.7GB de RAM.

  Los flags de compilación para la versión secuencial son:

  ```
  -O4 -mcmodel=large
  ```

Cabe citar que para las medidas utilizamos tiempo absoluto.

13.3. Aplicación sintética

Para medir esta aplicación daremos el tiempo medio de ejecución de tres ejecuciones en las tres de versiones disponibles: la versión secuencial, la versión especulativa con el motor origi-

	Tiempo en segundos de cada versión		
Número de threads	Secuencial	Especulativa original	Especulativa nuevo motor
1	0.022148	0.44719	1.309085
2	no procede	0.154503	2.101930
3	no procede	0.155107	2.109030
4	no procede	0.172052	2.643528

Cuadro 13.1: Resultados de la ejecución de la aplicación sintética en el servidor *Nodoyuna*.

	Tiempo en segundos de cada versión		
Número de threads	Secuencial	Especulativa original	Especulativa nuevo motor
1	0.214141	0.871557	13.012551
2	no procede	0.454997	13.225125
3	no procede	0.326118	9.386746
4	no procede	0.257993	8.018528

Cuadro 13.2: Resultados de la ejecución de la aplicación que calcula el menor círculo contenedor de una nube de puntos en el servidor *Nodoyuna*.

nal, y la versión especulativa con el nuevo motor desarrollado. Para las versiones especulativas tomaremos un tamaño de bloque de 1 000 iteraciones.

13.3.1. Resultados

Los resultados se pueden ver en el cuadro 13.1.

13.3.2. Conclusión parcial

Como se puede ver, esta aplicación da peores resultados para las versiones especulativas, esto está provocado porque se producen multitud de violaciones de dependencia, debido a que esta aplicación se utiliza para someter a cargas pesadas al motor, en busca de fallos.

13.4. Menor círculo contenedor

La nube de puntos con la que se contará para esta aplicación contará con una muestra de 10 000 000 puntos. Para realizar las medidas pertinentes a esta aplicación daremos el tiempo medio de ejecución de tres ejecuciones en las tres de versiones disponibles: la versión secuencial, la versión especulativa con el motor original, y la versión especulativa con el nuevo motor desarrollado. Para las versiones especulativas tomaremos un tamaño de bloque de 11 000 iteraciones.

13.4.1. Resultados

Los resultados se pueden ver en el cuadro 13.2.

	Tiempo en segundos de cada versión		
Número de threads	Secuencial	Especulativa original	Especulativa nuevo motor
1	0.022148	0.004671	178.688916
2	no procede	0.004463	311.332632
3	no procede	0.027800	639.354386
4	no procede	0.004038	837.456309

Cuadro 13.3: Resultados de la ejecución de la aplicación que calcula el cierre convexo de una nube de puntos en el servidor *Nodoyuna*.

13.4.2. Conclusión parcial

Para esta aplicación vemos que los resultados son mejores en la aplicación secuencial, sin embargo, a la vista de las mejoras obtenidas al aumentar el número de threads, cabría esperar que un aumento en el número de procesadores disponibles, provocase una disminución considerable en el tiempo de ejecución de las versiones especulativas, pudiendo alcanzar tiempos menores que el obtenido en la versión secuencial.

13.5. Envolvente convexo

La nube de puntos con la que se contará para esta aplicación contará con una muestra de 10 000 puntos. Para realizar las medidas pertinentes a esta aplicación daremos el tiempo medio de ejecución de tres ejecuciones en las tres de versiones disponibles: la versión secuencial, la versión especulativa con el motor original, y la versión especulativa con el nuevo motor desarrollado. Para las versiones especulativas tomaremos un tamaño de bloque de 2 500 iteraciones.

13.5.1. Resultados

Los resultados se pueden ver en el cuadro 13.3.

13.5.2. Conclusión parcial

En este caso se puede comprobar que los tiempos de ejecución de la versión paralela con el motor original, son menores que los tiempos de ejecución de la versión secuencial. Por otro lado, también podemos observar que la aplicación paralela con el nuevo motor es más lenta cuantos más procesadores asignemos. El principal motivo de ese incremento en los tiempos de ejecución lo provocan las búsquedas secuenciales. En esta aplicación es muy significativo porque el número de variables a contemplar es elevado (aprox. 27 000), mientras que en las anteriores no se apreciaba porque el número de variables especulativas era relativamente pequeño (100 en el caso de la aplicación sintética y 10 en el caso del menor círculo contenedor). Luego hemos podido comprobar que las búsquedas elemento a elemento en el nuevo motor son muy ineficientes.

13.6. Valoración general de los resultados

Los resultados experimentales en términos de tiempo de ejecución muestran claramente que el nuevo motor es más lento que el antiguo. Esto se debe a tres motivos:

- El primero radica en la complejidad inherente a este desarrollo: aunque el nuevo motor funciona correctamente, estamos comparando el rendimiento de un motor que acaba de desarrollarse con el de un motor que lleva diez años de continuas mejoras y optimizaciones. Estamos seguros de que aún queda margen de maniobra para una mejora sustancial en su rendimiento.

- En segundo lugar, el nuevo motor está diseñado para el caso general de la ejecución especulativa sobre estructuras de datos arbitrarias, no para un caso particular sobre vectores como el motor anterior. En consecuencia, esta generalidad se ha obtenido a costa de una pérdida de rendimiento en casos concretos.

- Finalmente, el nuevo motor se ve penalizado por las búsquedas realizadas en las operaciones de lectura y escritura especulativa. En las lecturas, el motor original simplemente comprobaba si el dato i-ésimo había sido usado por cada predecesor o no, lo que suponía una operación de orden constante. El nuevo motor, para hacer lo mismo, necesita buscar en *todos* los datos de cada uno de los predecesores. Algo similar sucede con las operaciones de escritura. Este problema supone una penalización enorme en el rendimiento. Como trabajo futuro se propone modificar las estructuras de datos usadas internamente por el motor, reemplazando las listas con los datos utilizados por cada procesador por estructuras que permitan búsquedas mucho más rápidas, como árboles.

Capítulo 14

Conclusiones

En este capítulo se detallan las conclusiones a las que se llega con los resultados obtenidos. Además se proporcionan una serie de posibles vías de continuación del trabajo realizado.

Este TFG ha consistido en lo siguiente:

- Se ha dado una introducción a la paralelización especulativa, siendo ésta, una técnica novedosa que proporciona una mejora de rendimiento en muchas aplicaciones. Además se han mostrado algunas de las diferentes vías de investigación que lleva a cabo el grupo *Trasgo* en este tema, siendo el motor *SpecEngine* original un trabajo realizado por miembros de este grupo.
- Se ha profundizado en el funcionamiento del motor especulativo original, dando una descripción exhaustiva de las estructuras que lo forman, así como de las operaciones que realiza. También se ha incluido un pequeño ejemplo de paralelización de un código de ejemplo.
- Se han detallado las limitaciones de que consta la arquitectura original del motor y se ha intentado proporcionar varias soluciones capaces de solventarlas.
- Se ha profundizado en los modelos teóricos de cada arquitectura propuesta, estudiando las ventajas, inconvenientes y costes de cada una de ellas, escogiendo la que, bajo nuestro punto de vista, era mejor.
- Se ha comentado cómo se llevó a cabo la implementación de la nueva arquitectura del motor para la solución seleccionada.
- Se ha elaborado una batería de pruebas para comprobar la corrección del código desarrollado.
- Se han adaptado a la nueva arquitectura del motor unos benchmarks de prueba de los que disponíamos en la versión original, con el objetivo de comparar los resultados obtenidos en ambas versiones del motor especulativo.

14.1. Comparativa de ambas versiones del motor

El principal inconveniente de la nueva solución es que las operaciones de lectura y escritura especulativas serán más lentas, porque no se conoce la posición o existencia de los datos en la matriz de cada thread, y deben realizarse búsquedas elemento a elemento. En la versión original del motor, los elementos ocupaban siempre la misma posición en el vector de cada thread.

Por otra parte, las ventajas de la nueva versión son las siguientes:

- Todas las aplicaciones que soportasen el motor existente serán compatibles con el nuevo, tras realizar unos pequeños cambios en la interfaz de las operaciones.
- La nueva versión del motor soporta aritmética de punteros, lo que permite, entre otras cosas, especular sobre datos en memoria dinámica o sobre variables sueltas, independientemente de la estructura a la que pertenezcan.
- Ahora podemos especular con diferentes tipos de datos, ya que el nuevo motor obtiene automáticamente su tamaño, tanto de datos escalares (char, int, double, etc.) como de datos más complejos.

14.2. Conclusiones

Tras realizar este TFG, se pueden extraer las siguientes conclusiones:

- La paralelización especulativa no siempre provoca una reducción en el tiempo de ejecución de las aplicaciones.
- La paralelización especulativa produce una ganancia significativa de aceleración cuando el número de procesadores disponibles es grande y durante la ejecución del programa no se producen demasiadas violaciones de dependencia.
- Es posible utilizar aritmética de punteros en la paralelización especulativa.
- Es posible paralelizar especulativamente aplicaciones que especulen sobre variables con diferente tipo de datos.
- La nueva versión del motor tiene más funcionalidad que la original, pero, en general, genera unos tiempos de ejecución mayores.

14.3. Trabajo futuro

- Reemplazar la matriz de punteros por una estructura de datos que no obligue a recorrer toda la matriz en busca de un dato, al objeto de mejorar sustancialmente los tiempos de las escrituras y lecturas especulativas.
- Implementar las operaciones de suma y cálculo del máximo, existentes en el motor original, pero no implementadas en la nueva arquitectura desarrollada.
- Mejorar la nueva versión del motor de forma que aumente el tamaño de las matrices dinámicamente, tal y como se planteó inicialmente.

- Implementar una nueva versión del motor que soporte cambios en el tamaño de los datos, es decir, que permitiese, por ejemplo, que un dato que inicialmente tenía un tamaño de dos bytes fuese sustituido por otro con tan sólo un byte.
- Implementar una optimización en el nuevo motor que, en lugar de almacenar siempre direcciones de memoria (lo que implica almacenar siempre enteros de 4 u 8 bytes), guarde las direcciones de memoria sólo si el dato referenciado es mayor que el tamaño del puntero, y que guarde simplemente el dato en caso contrario.

Este Trabajo de Fin de Grado ha sido aceptado para su presentación en las XXIII Jornadas de Paralelismo, que se celebrarán en Elche en septiembre del 2012.

Apéndice A

Contenido del CD-ROM

La memoria presente contiene adjunto un CD-ROM que incluye toda la información detallada del proyecto. En esta sección explicaremos el contenido de dicho CD-ROM y como está organizado. El CD-ROM contendrá el código utilizado en el desarrollo del sistema, tanto la nueva arquitectura propuesta, como los benchmarks adaptados a la misma, además contiene una copia digital de la memoria.

La estructura del CD-ROM se divide en dos carpetas, el contenido de cada una de ellas se corresponde con los contenidos principales del proyecto.

La carpeta "**Documentacion**" incluye la Memoria del Trabajo de Fin de Grado en formato digital, en concreto, en formato PDF.

En la carpeta "**Codigo_fuente**" se encuentra el código desarrollado para la construcción del proyecto, dentro de esta carpeta se encuentran dos subcarpetas que contienen el código desarrollado con la siguiente distribución:

- **Motor_especulativo_v10**: Contiene la nueva versión del motor especulativo (Tan sólo las librerías de funciones que la implementan).
- **Benchmarks**: Contiene otras tres subcarpetas cuyos contenidos son los siguientes:
  - **Aplicacion_sintetica**: Aquí se encuentra el código de la versión compatible con el nuevo motor especulativo de la aplicación sintética.
  - **Menor_circulo_contenedor**: En esta carpeta están incluídos los ficheros utilizados en el desarrollo de la versión para el nuevo motor de paralelización especulativa, de la aplicación que calcula el menor círculo contenedor de una nube de puntos en dos dimensiones.
  - **Envolvente_convexo**: Esta carpeta contiene los archivos desarrollados para implementar la aplicación que calcula el cierre convexo de una nube de puntos en dos dimensiones, mediante la nueva versión del motor especulativo.

Bibliografía

[1] José Manuel Mira Antonio Pallarés Ruiz Salvador Snachez Guillén Bernardo Cascales Salinas, Pascual Lucas Saolín. *LATEX Una imprenta en sus manos.* ADI, Martín de los heros, 66 28088 Madrid, 2005.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] Álvaro García Yágüez, Diego R. Llanos, and Arturo González-Escribano. Migración de un motor software de ejecución especulativa y evaluación de su rendimiento. Proyecto de fin de carrera, Universidad de Valladolid, July 2009.

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

[9] 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.

[10] Diego R. Llanos. Introducción a las técnicas de ejecución especulativa. In *Conferencias de paralelización especulativa en tiempo de ejecución*, October 2008.

[11] Diego R. Llanos. Un modelo software de ejecución especulativa. In *Conferencias de paralelización especulativa en tiempo de ejecución*, October 2008.

[12] Diego R. Llanos. *Fundamentos de Informática y Programación en C*. Paraninfo, Spain, 2010. ISBN: 978-84-9732-792-3.

[13] 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.

[14] 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.

[15] Roger S. Pressman. *Ingeniería del software, Un enfoque práctico*. McGraw Hill, 5th edition, 2002.

[16] Ian Sommerville. *Ingeniería del software*. Addison-Wesley, 7th edition, 2005. ISBN 84-7829-074-5.