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

Alvaro Estebanez, Diego R. Llanos y Arturo Gonzalez-Escribano[1]

*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 hardware o software monitoriza la ejecución y corrige eventuales violaciones de dependencia. A lo largo de este artículo se mostrará un mecanismo software para llevar a cabo la paralelización citada, en concreto, una extensión del motor de paralelización especulativa desarrollado por Cintra y Llanos, que solventa algunas de sus limitaciones actuales.**

*Palabras clave*— **paralelización especulativa, thread-level speculation, parallelism.**

## I. Introducción

La paralelización especulativa [1] 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. El procedimiento que sigue esta técnica consiste en ejecutar de manera optimista el bucle en paralelo, esperando que no se viole la semántica secuencial del bucle. A su vez, un sistema hardware o software monitoriza la ejecución de cada iteración procesada en paralelo, para detectar y corregir las violaciones de dependencia que puedan aparecer en el proceso.

### A. Herramienta de paralelización especulativa

En nuestro caso utilizamos el motor software de paralelización especulativa desarrollado por Cintra y Llanos [2], [3], [4], [5]. El motor especulativo pretende servir para ejecutar bucles de forma paralela en los que se desconoce la dependencia entre sus datos. Este motor se basa en un conjunto de librerías de funciones que permiten ejecutar especulativamente un bucle.

Procedamos a ver un ejemplo de cómo se ejecutaría especulativamente un bucle con la herramienta mencionada. Para ello nos basaremos en el bucle de la Fig. 1, con 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 considerarse como thread no especulativo, y el que corresponde a la última, será considerado como thread más especulativo. Para realizar su cometido, cada thread mantiene su versión de los datos compartidos, por tanto, si cada uno consolida sus resultados siguiendo un orden aleatorio, pueden aparecer incoherencias al finalizar el bucle. Es por eso por lo que cada thread debe consolidar sus datos de modo ordenado.

[1]Dpto. de Informática, Univ. de Valladolid, Spain, e-mail: alvaro.estebanez@alumnos.uva.es, {diego|arturo}@infor.uva.es

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

Fig. 1. Este tipo de bucle puede dar lugar a violaciones de dependencia en su ejecución paralela

Para comprender mejor los datos expuestos mostraremos un ejemplo de ejecución en la Fig. 2. En esta figura se puede apreciar una posible ejecución del bucle del ejemplo de modo paralelo. Se puede ver que todas las operaciones se realizan manteniendo la semántica secuencial, 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 hasta ese momento por el thread cuatro.

### B. Cambios necesarios en el código de las aplicaciones

Para evitar fallos indeseados, se necesita hacer una serie de modificaciones en el código original:

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.

### C. Limitaciones

Como se ha mencionado al comienzo del presente documento, la versión original del motor cuenta con una serie de restricciones que limitan su utilización. A continuación citaremos alguna de ellas:

- Sólo podemos trabajar con una estructura: La versión original del motor solamente permite la

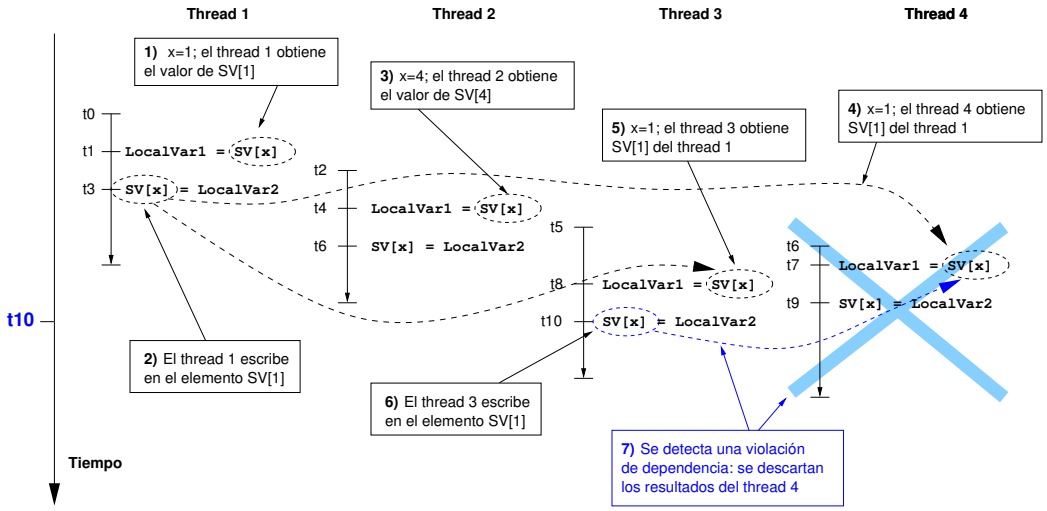


Fig. 2. En cada lectura se busca la versión más actualizada de `SV[X]`, consultando a los threads anteriores hasta llegar, en su caso, al thread no especulativo. En cada escritura se comprueba si un thread posterior ha utilizado un valor incorrecto, en cuyo caso se descarta su ejecución.

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 original no nos lo permite.
- El motor 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.

El resto de artículo se estructura como sigue. La sección II explica la estructura del motor original. La sección III describe la nueva arquitectura propuesta. La sección IV muestra cómo se realizaría una lectura especulativa con el nuevo motor. La sección V muestra, a su vez, el funcionamiento de la nueva escritura especulativa. La sección VI discute las ventajas e inconvenientes de ambas soluciones. Finalmente, la sección VII concluye este artículo.

## II. Arquitectura del SpecEngine 2003

El motor especulativo trata de evitar violaciones de dependencia, para lo cual utiliza ciertas estructuras de datos que se encargan de constatar las operaciones a las que están sometidas las variables especulativas en cada operación.

Como ya se ha dicho, cada thread deberá poseer su propia versión de los datos especulativos, por tanto, necesitamos un vector de datos por cada thread, además de un vector global donde constatar los datos una vez terminada la ejecución. Por otro lado, necesitamos una estructura de datos que mantenga el estado de ejecución de todos los threads. De ello se encargará otro vector que actuará como una ventana deslizante, con un tamaño mayor o igual al número de procesadores.

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. A medida que cada thread realiza sus operaciones y finaliza, la ventana de trabajo se desplaza a la derecha. Además de esto, la ventana contendrá dos apuntadores, uno al primer hueco en uso, también denominado no especulativo, y otro al último, el más especulativo.

Hemos visto que cada hueco se asigna a un procesador, y tiene asociada una versión de las variables especulativas. Además de conocer el valor de los datos propio a su thread, se debe conocer el estado de cada variable especulativa, 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. En esta matriz cada posición contendrá el estado de la versión de cada variable. Podemos ver un esquema de la arquitectura descrita anteriormente en la Fig. 3.

### A. Operaciones de lectura especulativa

Para ver cómo se lleva a cabo la lectura especulativa en el motor original nos basaremos en la Fig. 4.

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

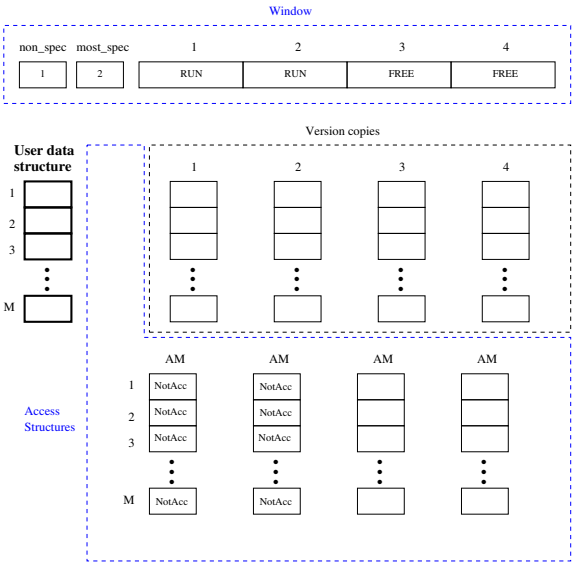


Fig. 3. Arquitectura original del motor especulativo

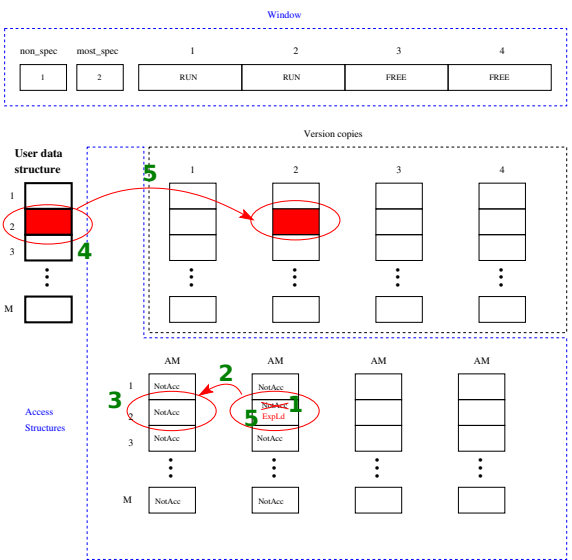


Fig. 4. Motor especulativo original: operación de lectura

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].

### B. Operaciones de escritura especulativa

Para ver cómo son las escrituras especulativas en el motor original nos basaremos en la Fig. 5.

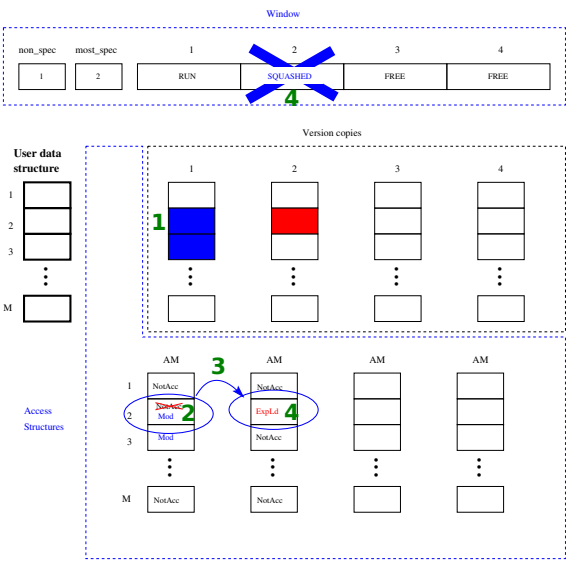


Fig. 5. Motor especulativo original: operación de escritura

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 ExpLd-Mod).
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.

## III. Descripción de la nueva arquitectura propuesta

Como objeto de nuestra investigación pretendemos elaborar una versión nueva del motor que nos permite solventar las limitaciones del motor original (expuestas con anterioridad). Para lograr nuestro objetivo hemos tenido que modificar casi al completo la arquitectura existente. A continuación citaremos las estructuras de que consta nuestra nueva versión.

### A. 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 Fig. 6, pero además a continuación describiremos cada uno de los componentes por separado:

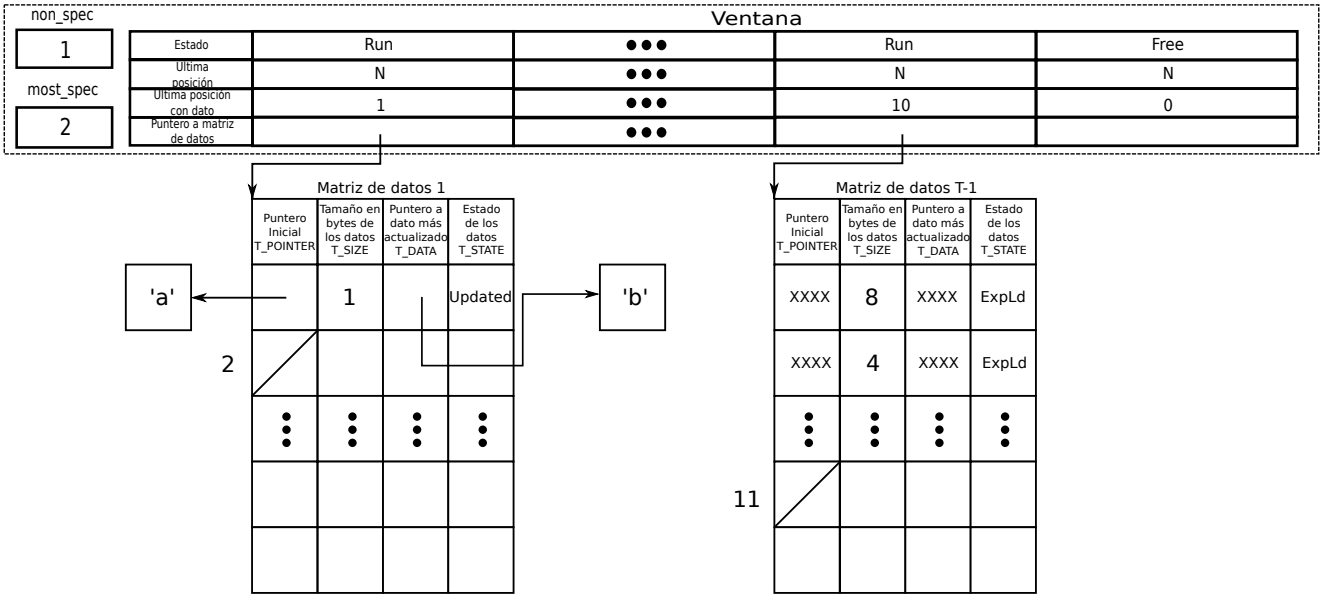


Fig. 6. Arquitectura del nuevo motor de paralelización especulativa.

- **Estado:** Como la versión anterior, es necesario conocer el estado de cada procesador, luego esta función es similar a la existente en la versión previa.
- **Puntero a matriz de datos:** La principal novedad ser 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.

### *B. 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 representación de la estructura en la Fig. 6.

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.

### *C. Apuntadores al thread no especulativo y al más especulativo*

Estos elementos tienen el mismo significado que en la versión inicial del motor, por tanto, se va a obviar su explicación.

En la Fig. 6 se pueden ver la nueva disposición del motor al completo, con las estructuras mencionadas.

## IV. Operaciones de lectura especulativa en el nuevo motor

Procederemos a explicar esta operación mediante un ejemplo, apoyándonos en imágenes. Supongamos que inicialmente tenemos en el motor los valores vistos en la Fig. 7(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 la Fig. 7(b).
2. Como no se ha encontrado el valor de la posición de memoria (sólo contiene los valores 1 000,

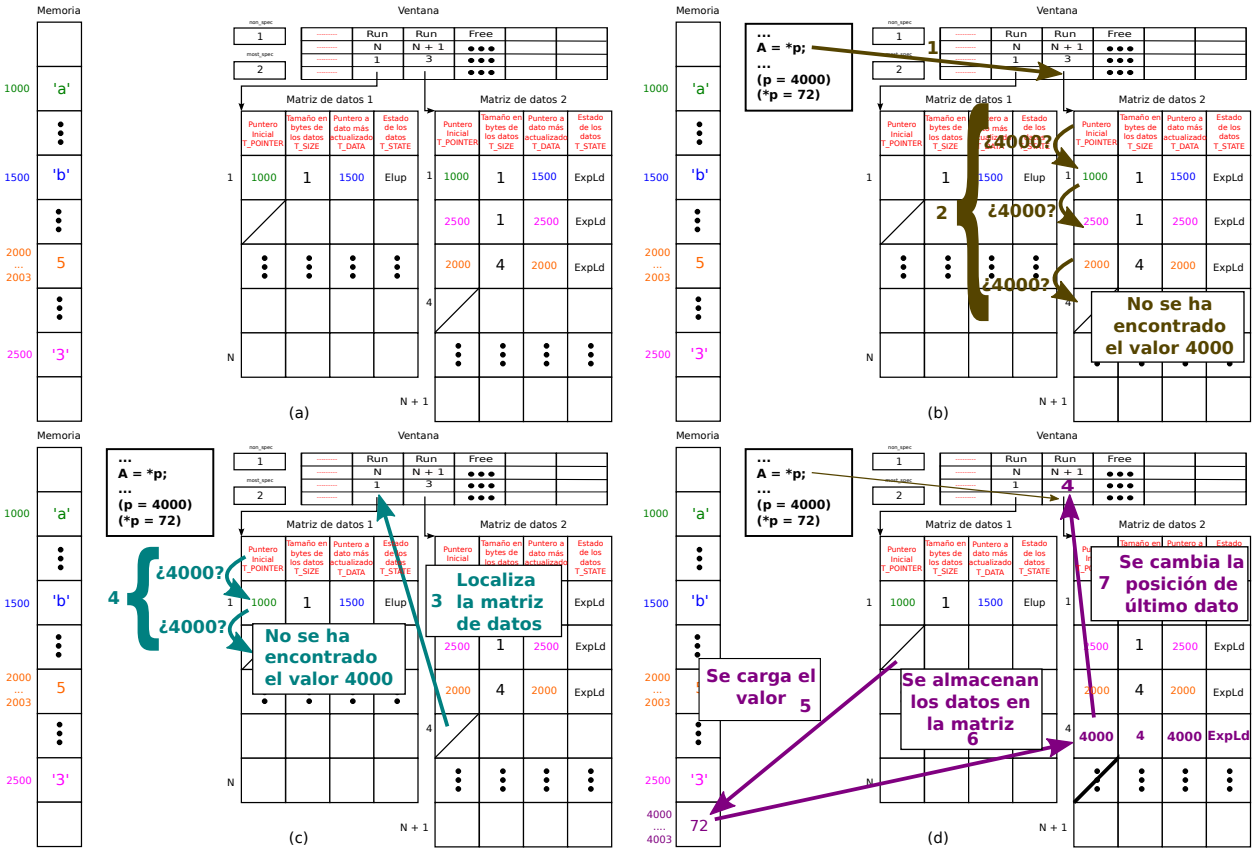


Fig. 7. Nuevo motor especulativo: operación de lectura.

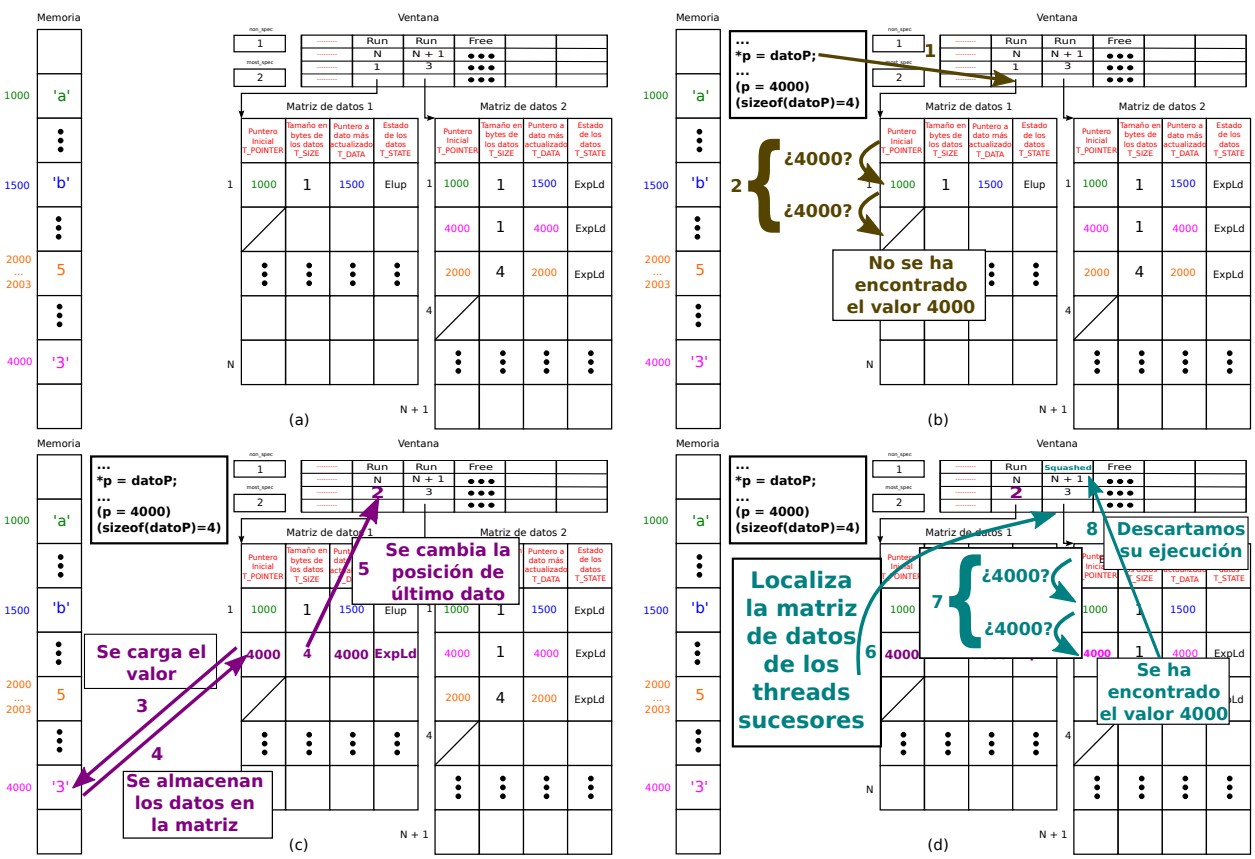


Fig. 8. Nuevo motor especulativo: operación de escritura.

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 Fig. 7(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 Fig. 7(d).

## V. Operaciones de escritura especulativa en el nuevo motor

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 Fig. 8(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 la Fig. 8(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 Fig. 8(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 Fig. 8(d).

## VI. Comparativa de ambas versiones

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.

## VII. Conclusiones

La paralelización especulativa permite explotar el paralelismo de ciertas aplicaciones. Para utilizar este tipo de paralelización es necesario un mecanismo que compruebe si se producen asignaciones fuera de orden. El grupo de investigación Trasgo utilizaba el motor `SpecEngine`, cuyo uso conlleva ciertas limitaciones, como por ejemplo la necesidad de especular sobre un sólo 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.

## Agradecimientos

Este trabajo ha sido financiado parcialmente por el Ministerio de Educación (Beca de Colaboración), el Ministerio de Industria (CENIT OCEANLIDER), Ministerio de Economía y programa FEDER de la Unión Europea (red CAPAP-H3 TIN2010-12011-E, TIN2011-25639, red CAPAP-H4 TIN2011-15734-E), y el proyecto HPC-EUROPA2 (project number: 228398), con el apoyo de la Comisión Europea (Capacities Area - Research Infrastructures Initiative).

## Referencias

[1] Arturo González-Escribano and Diego R. Llanos, "Paralelización especulativa y sus alternativas," *Actas XVIII Jornadas de Paralelismo*, 2007.
[2] 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.
[3] Marcelo Cintra and Diego R. Llanos, "Design space exploration of a software speculative parallelization scheme," *IEEE Trans. on Paral. and Distr. Systems*, vol. 16, no. 6, pp. 562–576, June 2005.
[4] Diego R. Llanos, "Introducción a las técnicas de ejecución especulativa," *Conferencias de paralelización especulativa en tiempo de ejecución*, October 2008.
[5] Diego R. Llanos, "Un modelo software de ejecución especulativa," *Conferencias de paralelización especulativa en tiempo de ejecución*, October 2008.