UNIVERSIDAD DE MURCIA

Facultad de Informática

Departamento de Ingeniería
y Tecnología de Computadores

Proyecto Fin de Carrera

El controlador de cerrojos:

Un mecanismo de sincronización eficiente en multiprocesadores ccNUMA

Murcia, Junio de 2006

Alumno:

José Rubén Titos Gil

Directores:

Manuel Eugenio Acacio Sánchez
José Manuel García Carrasco

Resumen

La sobrecarga introducida por las actividades de sincronización constituye un aspecto crítico en el rendimiento de los multiprocesadores de memoria compartida escalables. Los cerrojos son un mecanismo de sincronización empleado en los programas paralelos de memoria compartida, cuya finalidad es garantizar a los procesos el acceso en exclusión mutua a las secciones críticas del código en las que se modifican las variables compartidas del programa. La implementación eficiente de las operaciones de adquisición y liberación de cerrojos es crucial en estos sistemas. La elección de una solución inapropiada puede hacer de estas operaciones uno de los principales cuellos de botella, y por ello es fundamental encontrar el compromiso adecuado entre hardware y software a la hora de su implementación. Las técnicas de espera ocupada se usan muy a menudo para conseguir la exclusión mutua, dada su sencilla implementación en software y su buen rendimiento en circunstancias de poca competición por el cerrojo. Desafortunadamente, estas implementaciones tienden a producir contención en la memoria y la red de interconexión, un efecto que se hace notablemente más pronunciado conforme las aplicaciones escalan.

En este proyecto presentamos el *controlador de cerrojos*, una solución hardware capaz de lograr la sincronización eficiente en los multiprocesadores de memoria compartida escalables basados en directorio (ccNUMA). Con este mecanismo, se mantiene sencillo el algoritmo de adquisición, con las ventajas que esto conlleva en cerrojos con poca contención, mientras que el hardware se encarga de hacer eficiente la transferencia de aquellos otros cerrojos en los que un elevado número de procesadores compite por su acceso. Se trata de una lógica adicional acoplada al controlador de directorio que detecta dinámicamente el nivel de contención de cada cerrojo y aplica selectivamente una política de encolado de los peticionarios del cerrojo. Gracias a esto, el coste de la transferencia del cerrojo se hace constante e independiente del número de peticionarios, y además se consigue que estas actividades de sincronización no se vean afectadas por la elevada latencia de acceso a memoria.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En la historia de la Informática, diversas disciplinas científicas como la Física o la Biología han impulsado el desarrollo de la computación e influenciado claramente su evolución. Desde los orígenes del ordenador, la Ciencia ha explotado su extraordinaria capacidad de cálculo con el fin de resolver problemas muy complejos, sencillamente imposibles de afrontar de manera no automatizada.

En general, el conocimiento científico de una disciplina se aplica en la resolución de problemas: desde la predicción del tiempo atmosférico al control de reacciones químicas. Las leyes, métodos, etc. propios de cada área sientan la base sobre la que se construyen *algoritmos* susceptibles de ser implementados y ejecutados en un computador. Sin embargo, existen hoy día muchos problemas de ámbito científico cuya resolución con computadores convencionales no se puede completar en un tiempo finito. Pongamos como ejemplo un problema en Aeronáutica como es simular la aerodinámica del ala de un avión. Con la tecnología actual, un computador uniprocesador cuyas prestaciones son del orden del gigaflop/s (mil millones de operaciones en punto flotante por segundo) necesitaría unos 32 años para resolverlo.

Relajar las restricciones del problema o disminuir la precisión con la que se modela la realidad puede acortar ostensiblemente el tiempo de ejecución y los requisitos de memoria, pero esto conduce a resultados imprecisos y de escasa utilidad. Arrojar dichos programas al fondo de un cajón hasta que la tecnología satisfaga el rendimiento requerido es una alternativa. La otra, encontrar soluciones para ofrecer en el presente el rendimiento de las máquinas del futuro. Este es el reto de los arquitectos de computadores y la clave de su éxito se resume en una sola palabra: *paralelismo*.

El constante incremento en las prestaciones de los computadores en las cuatro últimas décadas, tal como pronosticó la *Ley de Moore*, no sólo está motivado por el avance en la integración de circuitos. El arquitecto de computadores juega un papel fundamental en este proceso, persiguiendo la organización hardware del sistema que extraiga el máximo partido de los componentes subyacentes y explote el paralelismo de unos programas cuya naturaleza es inherentemente secuencial.

Dentro del mundo de los sistemas uniprocesador, existen múltiples tipos de paralelismo: *paralelismo a nivel de bit*, *paralelismo a nivel de instrucción* (ILP) y *paralelismo a nivel de hilo* (TLP). La idea detrás del ILP es encontrar instrucciones que no dependan del resultado de otras instrucciones todavía por ejecutar, para ejecutarlas concurrentemente. Mediante la segmentación o *pipelining* de las etapas de procesamiento se consigue extraer el paralelismo temporal: el procesador alberga varias instrucciones al mismo tiempo, cada una en una fase de ejecución (decodificación, emisión, etc.). Los procesadores *superescalares* van un paso más lejos al explotar, además, el paralelismo espacial: éstos son capaces de decodificar, lanzar a ejecución y completar más de una instrucción en cada ciclo de reloj. Otra forma de paralelismo, denominado *intra-instrucción*, se basa en utilizar o bien palabras de instrucción muy largas (VLIW) o instrucciones SIMD (única instrucción-múltiples datos).

No obstante, el grado de paralelismo a nivel de instrucción (ILP) existente en los programas es limitado [Wal91]. Extender la ventana de instrucciones es la opción más inmediata, pero resulta impráctica ya que la lógica de detección de dependencias se hace prohibitivamente compleja. En su lugar, los arquitectos buscan más paralelismo contemplando simultáneamente la ejecución de más de un hilo o *thread*. La idea es la misma que en ILP, salvo que el paralelismo a nivel de hilo (TLP) examina instrucciones procedentes de diferentes hilos, en lugar de uno sólo.

A pesar de todas estas técnicas de extracción de paralelismo, el rendimiento ofrecido por un sistema uniprocesador es limitado y está condicionado en última instancia por la tecnología disponible en el momento de su fabricación. Aplicaciones científicas como el estudio del plegamiento de proteínas, el diseño aerodinámico en túneles de viento o el análisis del daño debido a impactos, y aplicaciones comerciales que van desde procesamiento de transacciones *on-line* a la minería de datos, requieren unas prestaciones muy por encima de las ofrecidas por un sistema uniprocesador.

Un sistema paralelo compuesto de múltiples procesadores tiene el potencial de superar las limitaciones de rendimiento de las arquitecturas uniprocesador. En el ejemplo anterior, sobre la simulación de la aerodinámica del ala de un avión, dijimos que tardaría 32 años en ser resuelto por un procesador actual. Ahora bien, si lográsemos que muchos procesadores como el anterior colaborasen en la resolución conjunta del problema, sería posible reducir este tiempo de manera considerable. Así, un supercomputador con una potencia de 1 teraflop (un billón de operaciones en punto flotante por segundo) resolvería el experimento en unas pocas semanas. La figura 1.1 ofrece una idea de la relación entre la potencia de cálculo y el tiempo necesario para resolver problemas de diversa complejidad.

Un multiprocesador con semejante potencia de cálculo está formado por cientos o miles de procesadores convencionales cuyo rendimiento individual es de unos pocos gigaflops. El problema está en cómo orquestar esos procesadores en la ejecución del problema. Idealmente, todos ellos realizan sólo cálculo útil y en su conjunto ejecutan las mismas instrucciones que un uniprocesador ejecutaría, sin que ninguno haga trabajo duplicado ni esté desocupado. Sin embargo, alcanzar este nivel de ejecución es

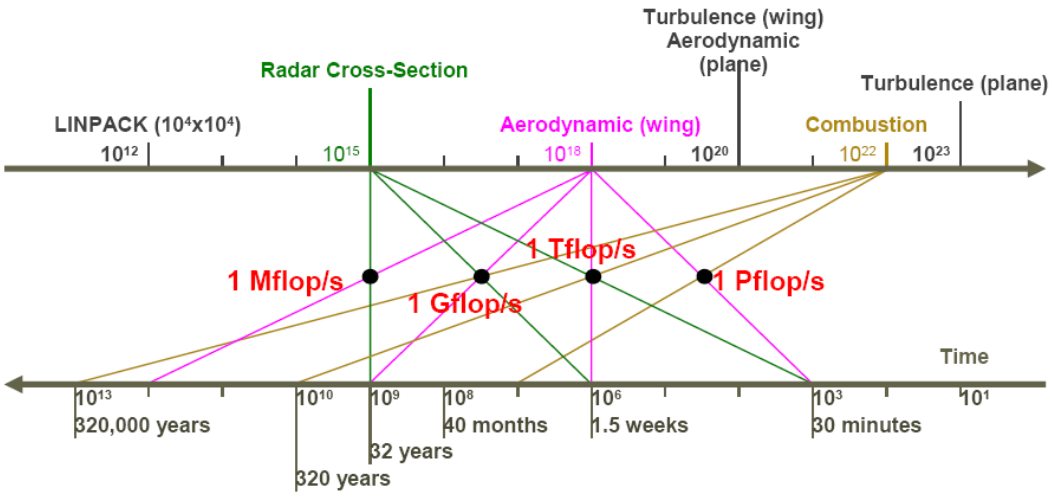

Figura 1.1: Número de operaciones vs. Tiempo de ejecución.

una meta utópica, debido a las distintas ineficiencias que surgen al utilizar múltiples procesadores en la resolución de un mismo problema

Uno de los factores que limita la eficiencia de un sistema paralelo es la coordinación y sincronización de las actividades llevadas a cabo por los distintos procesadores. Este trabajo se centra precisamente en las ineficiencias introducidas por dichas actividades de sincronización, si bien existen otros tipos, como son el desequilibrio de carga o los retardos de comunicación entre procesadores. Si bien conseguir el rendimiento ideal para un programa paralelo es una meta inalcanzable, no es necesario superar estas ineficiencias totalmente para hacer de los sistemas multiprocesador una alternativa competitiva en coste/prestaciones.

1.1. Arquitecturas paralelas

Un computador paralelo es en esencia un conjunto de elementos de procesamiento que se comunican y cooperan para resolver grandes problemas más rápidamente [AG89]. Normalmente, los elementos de procesamiento que se utilizan son microprocesadores. En un multiprocesador, los procesadores y módulos de memoria están conectados entre sí mediante una red de interconexión, a través de la cual los nodos intercambian datos mediante el envío de peticiones y la recepción de respuestas.

La comunicación entre los procesadores del sistema puede realizarse de varias maneras. Los dos paradigmas de programación paralela más comunes son el modelo de *paso de mensajes* y el de *memoria compartida*. Es importante distinguir entre el modelo de programación ofrecido (visión que el programador tiene de la máquina) y la plataforma hardware subyacente (implementación real de la máquina). En algunas ocasiones el mismo multiprocesador puede dar soporte a ambos modelos.

En las arquitecturas de *paso de mensajes* tradicionales, los procesadores se comunican entre sí accediendo directamente a la red de interconexión para enviar y recibir mensajes. Este paradigma permite al programador optimizar el movimiento de datos entre los nodos del sistema, así como usar las características de la red de manera óptima. Por contra, cualquier comunicación requerida entre los nodos del sistema ha de expresarse de manera explícita a través de operaciones como `SEND` y `RECEIVE`, lo cual dificulta la tarea del programador.

El modelo de *memoria compartida* resulta ser un paradigma de programación paralela más atractivo por su facilidad de programación, ofreciendo una abstracción del sistema intuitiva y a la vez permitiendo implementaciones razonablemente eficientes. La habilidad de un programador desarrollada en entornos uniprocesador sigue siendo válida. Este modelo soporta implícitamente la comunicación entre procesadores mediante instrucciones convencionales de carga y almacenamiento. El sistema de memoria subyacente es el encargado de atender la petición de un procesador para leer o escribir un dato en una posición de memoria remota, mediante el envío y recepción de mensajes al módulo de memoria adecuado en nombre de dicho procesador. Estas arquitecturas de memoria compartida son el eje sobre el que se basa este trabajo.

1.1.1. Multiprocesadores de memoria compartida

La mayoría de los sistemas de memoria compartida se valen de la replicación de datos en las cachés locales de cada procesador para reducir la latencia de los accesos a memoria, eliminado la necesidad de comunicación remota en cada acceso. No obstante, debido a presencia de múltiples copias del mismo bloque de memoria en la caché de los distintos procesadores, los multiprocesadores de memoria compartida necesitan utilizar un protocolo de coherencia de caché, con el fin de controlar que las modificaciones realizadas por un procesador en un bloque de memoria son observadas por el resto.

Cuando la red de interconexión que conecta los procesadores en el sistema es ordenada (tal como ocurre en un bus) la coherencia de caché se consigue mediante protocolos *snoopy*. Puesto que los mensajes que circulan por la red son observados por todos los procesadores simultáneamente, cada uno puede detectar aquellas transacciones sobre líneas que él tiene en estado modificado, y responder con la copia más reciente del bloque. Estas arquitecturas multiprocesador se denominan de acceso a memoria uniforme (UMA) o más comúnmente SMP's (*Symmetric MultiProcessors*), y su arquitectura básica se muestra en la figura 1.2 [CS99]. Puesto que el ancho de banda de memoria proporcionado por una red completamente ordenada es limitado, y cada procesador debe observar los fallos de caché del resto, las arquitecturas SMP son capaces de soportar configuraciones con hasta 64 procesadores.

Para escalar el número de procesadores de un sistema paralelo de memoria compartida a órdenes de cientos o miles es necesario que el ancho de banda de memoria

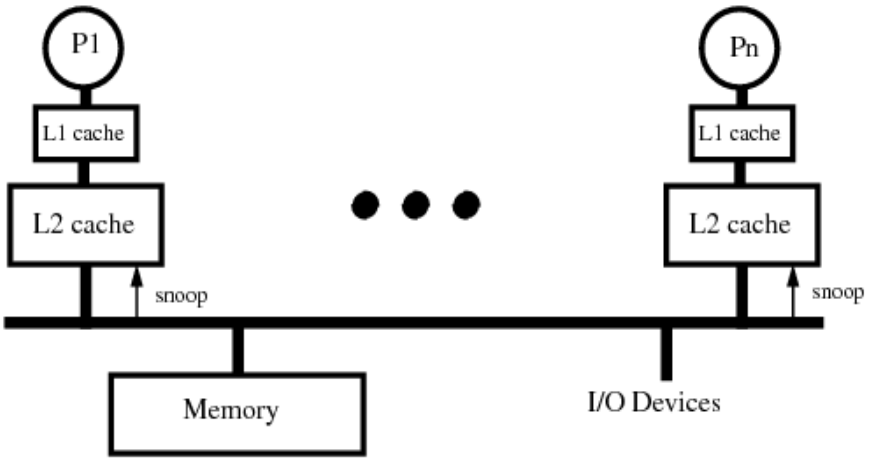

Figura 1.2: Arquitectura de un SMP con varios niveles de caché.

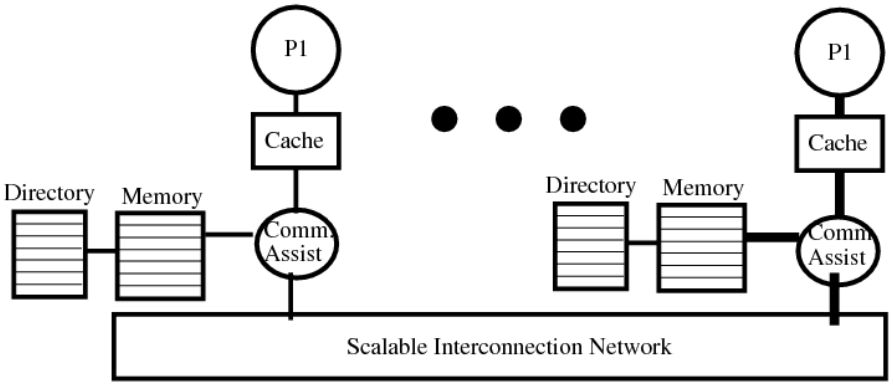

Figura 1.3: Arquitectura de un multiprocesador escalable basado en directorio.

escale conforme aumenta el tamaño del sistema. Esto se consigue sustituyendo la red completamente ordenada por una red de interconexión escalable punto a punto, de manera que la memoria está físicamente distribuida entre los diferentes procesadores del sistema. En estas máquinas, la agrupación de procesador y parte de la memoria total del sistema (además de las cachés y el controlador de memoria) es lo que se conoce como nodo. Los protocolos de coherencia de caché *snoopy* dejan de ser válidos al estar basado el sistema en una red no ordenada, pues deja de ser posible el que cada nodo fisgonee las transacciones de memoria del resto. Así surge una nueva estructura auxiliar en el sistema, denominada directorio, cuya función es esencialmente guardar información sobre el estado de cada bloque de memoria, en particular, qué nodos tienen copia del bloque en su caché y en qué estado. De esta forma, mediante el uso de un protocolo de coherencia basado en directorio es posible conseguir mantener la coherencia a nivel de caché en estos multiprocesadores escalables de memoria compartida distribuida (DSM, *Distributed Shared Memory*). La figura 1.3 [CS99] muestra un esquema de este tipo de arquitectura, también denominadas cc-NUMA (*cache coherent, Non Uniform Memory Access*).

1.2. Sincronización por medio de cerrojos

Un multiprocesador de memoria compartida ejecuta múltiples hilos de control, por lo general un hilo por procesador. En lo que respecta a nuestro trabajo, todos los hilos de control ejecutan el mismo programa y cada uno de ellos trabaja en una parte diferente del conjunto de datos. A este modelo de ejecución se le conoce como paradigma SPMD (*single program, multiple data streams*) y es un enfoque similar al esquema MIMD (*multiple program, multiple data streams*), salvo que en este último caso cada procesador ejecuta diferentes instrucciones que operan sobre datos diferentes. Aquí asumiremos que cada uno ejecuta una única instancia del programa paralelo, a la que llamaremos proceso. En cualquier caso, a partir de ahora se van a utilizar indistintamente los términos proceso y procesador.

La comunicación entre los procesos de un programa paralelo en un multiprocesador de memoria compartida tiene lugar de manera implícita a través de las instrucciones de carga y almacenamiento: un proceso escribe un nuevo valor en memoria y posteriormente otros procesadores lo leen. Por ello, la corrección de los programas paralelos depende de la habilidad de los procesos para observar esos nuevos valores en memoria en el momento apropiado. Cuando un proceso inicia una secuencia de cambios sobre una estructura de datos, ésta atraviesa un estado inconsistente de manera temporal, hasta que dicha secuencia se ha completado. Por tanto, otros procesos nunca deben intentar leer la estructura de datos mientras dicha transacción se está ejecutando, sino que estas modificaciones deben parecer ejecutarse de manera atómica.

Para conseguir dicha atomicidad, los programas paralelos protegen estas secciones críticas del código mediante variables de sincronización denominadas *cerrojos*. El funcionamiento de un cerrojo es muy simple: cuando un proceso lo adquiere, ningún otro proceso es capaz de adquirirlo hasta que el actual propietario lo libera. Así, los procesos que desean realizar una transacción atómica deben adquirir primero el correspondiente cerrojo. Una vez adquirido, el propietario tiene garantizado el acceso en exclusión mutua a la sección crítica, es decir, es seguro que no habrá otros procesos intentando acceder a las direcciones de memoria modificadas por dicha transacción. Cuando el propietario del cerrojo finaliza la transacción, otorga acceso al resto de procesos mediante la liberación del cerrojo.

Un problema que se ha de superar en la implementación de cerrojos es que la misma variable puede ser utilizada en distintas ocasiones bajo circunstancias de ejecución muy variadas y por tanto con diferentes requisitos de rendimiento. El mismo cerrojo puede ser accedido en un escenario de alta contención en el cual la mayoría de los procesos pasarán tiempo esperando la adquisición; en ese caso lo que se busca es un algoritmo que ofrezca alto ancho de banda de adquisición/liberación. Por otro lado, cuando sólo un procesador intenta acceder al cerrojo, la meta es ofrecer una latencia de adquisición reducida. Otros objetivos en la implementación de cerrojos son la escalabilidad, la sobrecarga de almacenamiento reducida y la imparcialidad en la concesión del cerrojo.

```
lock:  lw  Reg, cerrojo        unlock:  st  cerrojo, #0
       bnz Reg, lock                    ret
       t&s Reg, cerrojo
       bnz Reg, lock
       ret
```

Figura 1.4: Implementación de `LOCK()` y `UNLOCK()` mediante test&test&set.

A la hora de implementar las operaciones de adquisición y liberación de un cerrojo, cabe preguntarse qué roles deben jugar el usuario, el software de sistema y el hardware. En general, el programador quiere usar los cerrojos sin tener que preocuparse acerca de su implementación interna, así que la implementación se deja al sistema, que es quien decide el soporte hardware y la funcionalidad implementada en software.

Las instrucciones utilizadas para la implementación de los métodos de adquisición y liberación de cerrojos se denominan comúnmente *primitivas de sincronización*. Prácticamente todos los procesadores actuales proporcionan soporte hardware a la sincronización y cuentan en su repertorio de instrucciones con alguna primitiva de lectura-modificación-escritura (RMW) atómica, como por ejemplo test&set. Ésta lee el valor de una posición de memoria y almacena la constante 1 en esa posición si el valor leído es 0, todo ello de manera atómica. A partir de estas primitivas se implementa en software los métodos de adquisición y liberación de cerrojos.

1.2.1. Definición del problema

Nuestro estudio sobre las ineficiencias introducidas por las actividades de sincronización mediante cerrojos toma como punto de partida uno de las implementaciones más sencillas para proporcionar el método de adquisición: test&test&set. Esta implementación es una extensión directa de test&set, en la cual primero realiza una lectura del cerrojo para ver su estado y sólo cuando se observa que está libre, se intenta su adquisición ejecutando la operación de lectura-modificación-escritura atómica. En ciertas implementaciones, como ocurre en la librería del simulador RSIM, el método ejecuta en primer lugar test&set y en caso de fallo entonces pasa a la fase de espera ocupada (test). Con esto se consigue reducir la latencia de adquisición del cerrojo en situaciones de baja contención. En la figura 1.4 se muestra en pseudo-código ensamblador estilo MIPS las operaciones de adquisición y liberación con test&test&set.

Aunque se han propuesto numerosas construcciones para la implementación de cerrojos en software, la simplicidad y portabilidad de los cerrojos test&test&set hace de ellos una elección bastante popular. De hecho, a menudo los fabricantes de microprocesadores recomiendan en sus manuales hardware el uso de estos cerrojos sencillos como mecanismo portable de sincronización. Tal es el caso de las arquitecturas Alpha [DEC98], PowerPC [IBM98] o el MIPS R10000 [SGI98]. También se aconseja a los vendedores de bases de datos usar cerrojos simples en sus implementaciones [KSS96]. El estándar de hilos POSIX recomienda que la sincronización sea implementada me-

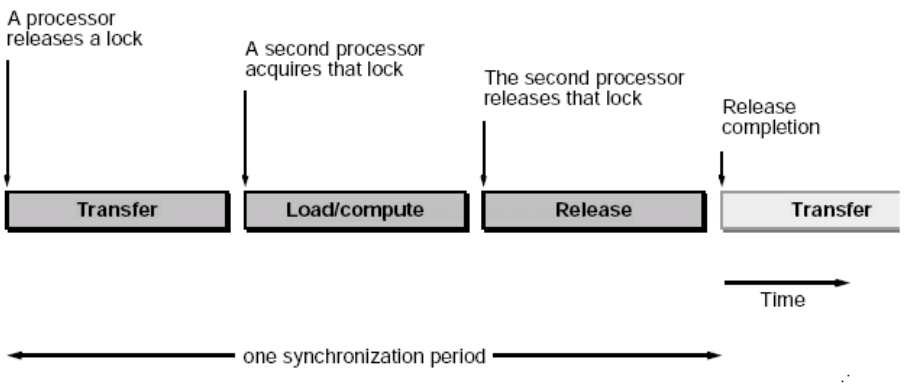

Figura 1.5: Periodo de sincronización.

diante llamadas a librerías como `pthread_mutex_lock()`, procedimientos éstos en los que la exclusión mutua se consigue usando cerrojos test&set y test&test&set [RG01].

El problema de la implementación test&test&set es el elevado tráfico que se genera en el acceso al candado en situaciones de elevada contención, es decir, cuando varios procesadores intentan adquirir simultáneamente el mismo cerrojo. Para conseguir una idea más clara del problema, veamos cómo se comporta test&test&set en un escenario de liberación-adquisición del cerrojo en el que participan tres procesadores. La situación descrita a continuación corresponde, en este orden, con las fases de liberación (*release*) y transferencia (*transfer*) que ocurren dentro de un periodo de sincronización, como muestra la figura 1.5 [Kä99].

Partimos de un estado inicial en el que el procesador P1 tiene el cerrojo en propiedad y está ejecutando la sección crítica, mientras que P2 y P3 están realizando espera activa sobre el cerrojo ocupado. A continuación se muestra la sucesión de mensajes necesarios hasta llegar al estado en el que P2 es propietario del cerrojo y está en la sección crítica y P3 permanece a la espera del candado. *Sharers* indica el estado de la entrada de directorio correspondiente al bloque de memoria del cerrojo, es decir, qué cachés tienen copia de dicho bloque en cada momento. A la derecha se muestra el número acumulado de mensajes generados hasta el momento.

- Situación inicial:

  P1 en sección crítica

  P2 y P3 espera activa Sharers(lock) = {P2,P3}

- P1 libera:

  P1 solicita ReadOwn a CD (1)

  CD envía invalidar a P2 y P3 (3)

  CD recibe inval.ACK de P2 y P3 (5)

  CD responde Upgrade a P1 (6)

  Sharers(lock) = {P1}. Se libera el cerrojo

- P2 y P3 fallan en caché al leer el cerrojo

  P1 y P2 envían ReadSh a CD (8)

  CD envía buscar a P1, destino P2 (9)

  P1 envía cache-to-cache a P2+copyback (11)

  CD responde ReadSh a P3 desde mem. (12)

  Sharers(lock) = {P1,P2,P3}

- P2 y P3 ven lock libre y ejecutan test&set

  P2 y P3 solicitan Upgrade a CD (14)

  Sharers(lock)={P1,P2,P3}

- Supuesto que primero llega el Upgrade de P2

  CD envía invalidar a P1 y P3 (16)

  CD recibe invl.ACK de P1 y P3 (18)

  CD responde Upgrade a P2 (19)

  P2 cierra cerrojo y entra en SC

  Sharers(lock)={P2}

- CD procesa después Upgrade de P3

  CD envía buscar/invalidar a P2, dest. P3 (20)

  P2 envía cache-to-cache a P3 + ACK (22)

  P3 ve el cerrojo ocupado

  P3 vuelve a la espera activa

  Sharers(lock)={P3}

- TOTAL: 22 mensajes / transferencia

Como vemos, la transferencia del cerrojo de P1 a P2 con tan sólo dos procesadores compitiendo por éste ha generado 22 transacciones en la red. En general, en una máquina de p procesadores, el método de test&test&set genera un tráfico de liberación del cerrojo que crece cuadráticamente con p. Este elevado tráfico de liberación generado por test&test&set en situaciones de contención introduce una gran ineficiencia en la sincronización mediante cerrojos en los programas paralelos de memoria compartida y constituye la base del problema que vamos a abordar en este trabajo.

1.2.2. Motivación y objetivo del proyecto

La proliferación de los multiprocesadores de memoria compartida distribuida y su amplia aceptación como plataformas viables para la computación paralela motivan la investigación del compromiso hardware/software en estos sistemas y la detección de

los cuellos de botella críticos en el rendimiento que pueden hacer imposible la escalabilidad de las aplicaciones. A pesar de la gran cantidad de trabajo relacionado que aparece en la literatura, la sobrecarga introducida por las actividades de sincronización es claramente un cuello de botella de los programas paralelos ejecutados sobre multiprocesadores de memoria compartida con coherencia de caché.

El objetivo de este trabajo es analizar cuantitativamente la repercusión que las operaciones de sincronización mediante cerrojos tienen en el rendimiento de un multiprocesador de memoria compartida basado en directorio, así como evaluar la mejora en las prestaciones que se consigue al diferenciar, a nivel del directorio, los accesos a las variables cerrojo del resto de variables compartidas (datos) del programa, utilizando dicha información adicional para adecuar la política de gestión de la línea de memoria que contiene el cerrojo.

Hasta ahora, el Grupo de Arquitectura y Computación Paralela (GACOP) de la Universidad de Murcia ha llevado a cabo diversos estudios y novedosas propuestas en el área de las arquitecturas cc-NUMA [AGGD01] [AGGD02] [AG03] [AGGD04] [AGGD05] [RAG05]. Este proyecto se enmarca en dicha línea de trabajo, como complemento a dichas aportaciones, y surge ante la necesidad de caracterizar la sobrecarga introducida por las actividades de sincronización.

1.2.3. Organización de este trabajo

El resto de este documento está organizado de la siguiente manera: En el capítulo 2 hacemos un breve repaso de los estudios más representativos acerca de sincronización basada en cerrojos. En el capítulo 3 mostramos algunos aspectos importantes sobre este tipo de sincronización y describimos el *controlador de cerrojos*, un mecanismo basado en directorio para conseguir la transferencia eficiente de cerrojos en situaciones de contención. El capítulo 4 describe el entorno de evaluación en el que hemos llevado a cabo los experimentos. En el capítulo 5 se muestran los resultados obtenidos en la evaluación de nuestra propuesta, y una breve caracterización de la sincronización en las aplicaciones bajo estudio. El capítulo 6 resume las principales conclusiones derivadas de este trabajo y sugiere algunas líneas de trabajo futuro que pueden ser exploradas en este tema.

Capítulo 2

Trabajo relacionado

Prácticamente todos los procesadores modernos ofrecen en su repertorio de instrucciones (ISA) alguna instrucción de lectura-modificación-escritura atómica para dar soporte de sincronización. A partir operaciones como `test&set`, `compare&swap`, `fetch&op` o el par LL/SC (carga enlazada y almacenamiento condicional) se construyen operaciones de más alto nivel como `LOCK()` y `UNLOCK()` que el programador utiliza directamente para garantizar el acceso en exclusión mutua a variables compartidas [CS99].

La implementación más simple del método de adquisición consiste en ejecutar test&set repetidamente mientras el valor devuelto no es 0. El problema es el tráfico generado, ya que cada ejecución de la instrucción test&set requiere la invalidación de la copia del anterior propietario.

Para prevenir que las limitaciones de la memoria y los recursos de comunicación se conviertan en cuellos de botella, se necesitan mecanismos de sincronización efectivos. Estas soluciones se pueden dividir según su naturaleza hardware o software.

2.1. Mecanismos de sincronización software

En el algoritmo de cerrojos *test&test&set* [RS84], los procesadores realizan la espera activa sobre la variable utilizando cargas convencionales en lugar de test&set, de modo que los accesos que se satisfacen en caché. Sólo cuando el cerrojo queda libre los procesadores ejecutan test&set para intentar adquirirlo. El tráfico tras la liberación es proporcional al cuadrado del número de procesadores en espera, así que en situaciones de mucha contención se hace muy elevado y por tanto no es escalable.

Para remediar este problema, se ha aplicado a este algoritmo la retirada o *backoff* [And90] [MCS91], cuya idea es insertar un retraso proporcional o exponencial tras cada intento fallido de adquisición. Con el retraso preciso se consigue reducir el tráfico

generado y mejorar el rendimiento, aunque el método sigue sin ser escalable.

Otra aproximación es *ticket lock* [MCS91], con el que usando dos contadores se evita que todos los procesadores a la espera traten de adquirir al mismo tiempo el cerrojo liberado. Al intentar adquirir el cerrojo, cada procesador obtiene su número con `fetch&increment` y espera comparando su ticket con el valor en servicio, hasta ambos valores son iguales y entra a la sección crítica. Todavía genera O(p) fallos de lectura tras cada liberación del cerrojo, debido a que todos esperan consultando la misma variable contador.

Los algoritmos de *cola basados en arrays* [And90] [GT90] consiguen que el coste de la transferencia del candado se reduzca a una invalidación y un fallo de lectura, O(1). En la adquisición se utiliza `fetch&increment` para obtener una dirección de memoria única -en vez de un valor- sobre la que realizar la espera activa, de manera que los accesos se satisfacen localmente. Se utiliza por cada cerrojo un array circular con p posiciones y los dos contadores de ticket lock. Su principal desventaja es la latencia de adquisición, siendo poco atractivo para condiciones de baja contención.

Otra forma de llevar a cabo la sincronización es utilizar un algoritmo que genera una única *lista enlazada* por cerrojo [MCS91] [MLH94]. Cada peticionario del cerrojo mantiene un registro con el flag en el que realizar la espera activa de manera local. Este registro está enlazado mediante un puntero con el siguiente peticionario. Para liberar al siguiente en la cola el propietario sólo tiene que activar su flag.

Por último, dentro de los mecanismos software se ha desarrollado un enfoque denominado *sincronización reactiva* [LA94], en el cual se utilizan diferentes algoritmos que se intercambian dinámicamente dependiendo del nivel de contención observado en cada momento.

2.2. Mecanismos de sincronización hardware

Las soluciones hardware mejoran sustancialmente en rendimiento a las alternativas desarrolladas en software, como se ha mostrado en varios estudios [HM93] [Kä99].

El mecanismo de *sincronización basado en caché* propuesto en [RL96] integra la sincronización en el protocolo de coherencia de caché para construir una cola de peticionarios del candado, añadiendo nuevos estados al controlador de caché. Otro trabajo previo que implementa una cola hardware en caché es QOLB [GVW89] [KBG97]. Su principal característica es que permite la entrega de los datos compartidos al mismo tiempo que se transfiere el candado (*colocación*), si bien los beneficios aportados dependen del tamaño de la línea de caché y los datos compartidos [Kä99]. Inspirado por QOLB, el Stanford DASH implementa la sincronización mediante colas a nivel del directorio [LLJ+92]. Esto conduce a un diseño más simple, pero también introduce un nivel de indirección, que incrementa el tiempo de transferencia.

Un enfoque diferente presentado en [HM93] es el uso de *memoria transaccional* (TM) como generalización de las primitivas LL/SC. TM se vale de instrucciones especiales para poder realizar accesos atómicos, no a una, sino a múltiples posiciones de memoria independientes, de forma que las secciones críticas ya no necesitan estar protegidas por variables cerrojo. El coste de TM es elevado: se necesita modificar el ISA y añadir una caché adicional para detección de conflictos y almacenamiento temporal de datos especulados.

Como en TM, la *elisión especulativa de cerrojos* (SLE) proporciona sincronización libre de cerrojos [RG01]. No obstante, si la especulación falla hay adquisición mediante el mecanismo tradicional, así que su rendimiento es pobre en presencia de conflictos. Es transparente al programador y no requiere instrucciones, caché y protocolo de coherencia especiales, pero sí la modificación del buffer de escritura y la adición de una unidad de detección de especulaciones erróneas.

La *eliminación de cerrojos transaccional* (TLR) es una extensión a SLE [Raj02]. En TLR las tareas se ejecutan especulativamente usando SLE; si se produce algún conflicto de datos, se resuelve dinámicamente usando marcas de tiempo (*timestamps*). El procesador con la marca más temprana confirma sus datos y los demás reinician su ejecución. Además del soporte requerido por SLE, TLR necesita una cola hardware para las peticiones y lógica adicional en el controlador de coherencia, pues añade un nuevo estado para distinguir los datos especulados con TLR.

La propuesta de *unidad de sincronización especulativa* (SSU) se aparta de la estrategia de sincronización libre de cerrojos [MT01]. Pese a que la adquisición del cerrojo puede causar efecto convoy en las tareas, SSU sí que garantiza el avance en la ejecución al contrario que SLE y TM. Además, se puede aprovechar de los beneficios de SLE, adoptando un enfoque de sincronización adaptativo, basada o libre cerrojos, dependiendo de la presencia o ausencia de conflictos, respectivamente [MT02]. SSU requiere modificaciones en la jerarquía de memoria, en forma de etiquetas, bits y líneas de caché extra para las variables de sincronización.

Otra propuesta, denominada *caché de cerrojos System-on-a-Chip* [ALM01] (SoCLC) utiliza una unidad hardware on-chip consistente en registros de un bit para almacenar las variables cerrojo, junto con la lógica de control asociada para implementar la sincronización vía interrupciones, en lugar de con espera ocupada. SoCLC no requiere instrucciones tipo LL/SC o test&set, ni protocolos de caché extendidos, es una solución independiente del núcleo.

La *reordenación de cerrojos especulativa* (SLR) [RS02] explota el hecho de que las secciones críticas protegidas por el mismo cerrojo se pueden ejecutar fuera de orden, con el fin de evitar reinicios innecesarios. SLR detecta las dependencias entre las secciones críticas ejecutadas especulativamente y después las reordena para minimizar el número de reinicios. Se implementa mediante un co-procesador dedicado a la especulación conectado al núcleo principal y las cachés, y una unidad generadora de secuencias de compleción para cada procesador.

Capítulo 3

Sincronización mediante cerrojos en multiprocesadores cc-NUMA

En este capítulo presentamos nuestra propuesta para optimizar las operaciones de sincronización mediante cerrojos en arquitecturas con coherencia de caché escalables basadas en directorio (cc-NUMA). En primer lugar mostramos cómo la sincronización sigue siendo un problema muy a tener en cuenta en las arquitecturas multiprocesador de memoria compartida del presente y futuro. A continuación, introducimos la idea de utilizar el directorio como controlador en la transferencia de cerrojos. Finalmente describimos en detalle el mecanismo que hemos desarrollado para aplicar la política de gestión de los bloques cerrojo a nivel de directorio: *el controlador de cerrojos*.

3.1. El problema de la sincronización

Diversos estudios en la última década han evaluado la influencia de la sincronización en el rendimiento de los programas paralelos [MCS91] [LA94] [KBG97] [Kä99]. Una de estas evaluaciones [KJCS99] concluye que la inclusión de soporte hardware para sincronización puede no tener una mejora significativa en el rendimiento de las aplicaciones, y por tanto apunta que dicha adición no justificaría el elevado coste de su implementación. No obstante, estudios posteriores han continuado sus esfuerzos en desarrollar mecanismos hardware que alivien el impacto de la sincronización en el rendimiento [RG01] [MT01] [Raj02] [RS02] [ALM01].

3.1.1. Impacto en el rendimiento

El continuo aumento de la diferencia entre la velocidad de la memoria y el procesador provoca que la influencia de la sincronización en el rendimiento global de las aplicaciones se haga más presente. Su peso en el tiempo total de ejecución aumenta

de manera proporcional a la distancia en términos de ciclos entre memoria y procesador, limitando la escalabilidad de las aplicaciones y su eficiencia. Durante muchos años, la tendencia dominante en los multiprocesadores de memoria compartida ha sido evitar el uso de hardware específico para sincronización e implementar en software estas operaciones, apoyándose únicamente en las primitivas atómicas proporcionadas por los microprocesadores. Pese a esta trayectoria, el *gap* procesador-memoria en las arquitecturas del futuro puede invertir dicha tendencia.

Uno de los objetivos que perseguimos en este proyecto es evaluar cuantitativamente la sobrecarga introducida por las actividades de sincronización y conocer en qué medida afecta la latencia de memoria a dicha sobrecarga, con el fin de plantear nuevas soluciones cuyo rendimiento sea independiente del *muro procesador-memoria*. En la actualidad un acceso toma entre los 80 y los 150 ciclos de procesador, dependiendo de varios factores como la tecnología de memoria y la frecuencia de procesador y bus. Este valor crece irremediablemente pese a la innovación tecnológica, y alrededor de 2010 se estima que alcanzará los 300 ciclos [BW04].

En nuestra caracterización del impacto de la sincronización mediante cerrojos tomamos como punto de partida el sencillo algoritmo de test&test&set, ya que éste es el método implementado en la librería del simulador RSIM. La decisión de partir de una implementación tan básica de la adquisición de cerrojos no es arbitraria, sino que está en la línea de trabajo de nuestro grupo de investigación: así, la mayoría de los trabajos previos del GACOP en torno a las arquitecturas cc-NUMA se han evaluado utilizando RSIM y por tanto utilizando cerrojos test&test&set.

En la figura 3.1 se muestra el tiempo de ejecución de cinco aplicaciones paralelas, obtenido mediante el simulador RSIM utilizando latencias de memoria de 80 y 300 ciclos, normalizado al caso de 300 ciclos. En estos gráficos hemos separado el tiempo de ejecución de cada aplicación en sus distintos componentes. De entre todos los componentes, estamos interesados en el que corresponde a la sincronización de cerrojo (*Acq*, *Rel*). A simple vista podemos observar en los gráficos cómo el número de ciclos dedicados a sincronización aumenta considerablemente al pasar de 80 a 300 ciclos de latencia de memoria, si bien su peso relativo al tiempo de ejecución total se mantiene similar en cada caso. La descripción del entorno de evaluación, la arquitectura simulada y las aplicaciones utilizadas para obtener estos datos preliminares se puede encontrar en los capítulos 4 y 5.

Dependiendo de la aplicación particular, la fracción del tiempo dedicado a la adquisición de cerrojos se sitúa entre el 10 y el 28 % cuando se necesitan 300 ciclos para acceder a memoria. Estos elevados porcentajes ponen de manifiesto que el mecanismo de sincronización utilizado en la arquitectura simulada por RSIM (test&test&set) se convierte en uno de los principales cuellos de botella cuando la latencia de la memoria se agranda. Así pues, es necesario prestar una atención especial a este aspecto y encontrar soluciones que atenúen esta ineficiencia. En este sentido, lograr un compromiso adecuado entre hardware y software es imprescindible para conseguir esta meta.

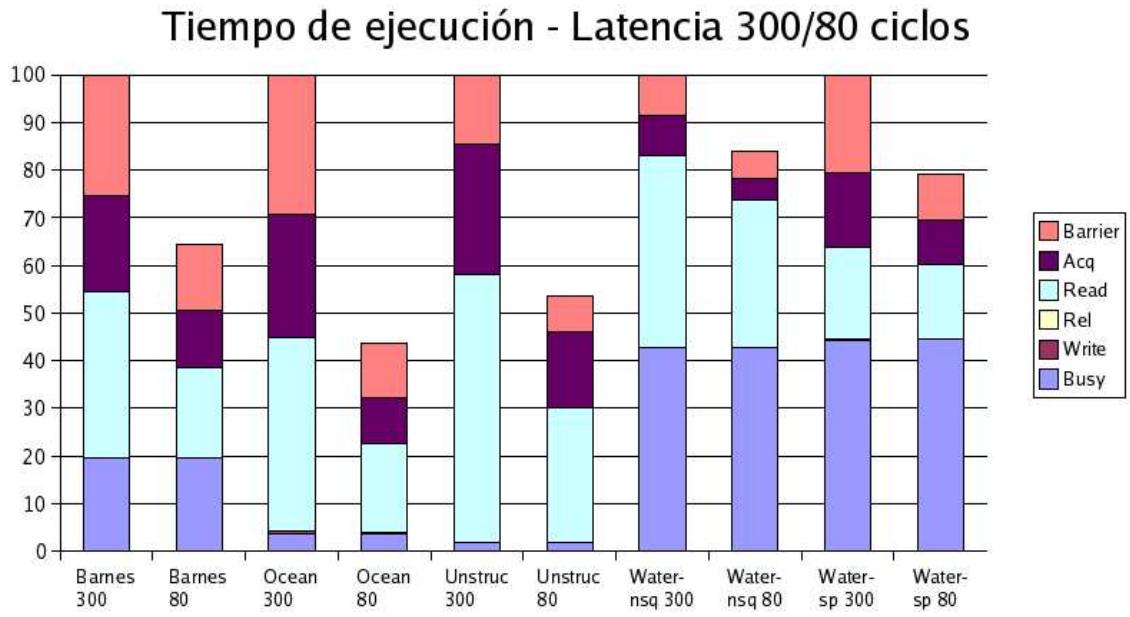

Figura 3.1: Tiempo de ejecución normalizado para cinco aplicaciones con latencia de memoria 80 y 300 ciclos.

3.1.2. Compromiso hardware/software

El modelo de programación de memoria compartida está actualmente establecido como el paradigma dominante en computación paralela [RKG04]. La abstracción de la memoria compartida facilita en gran medida la tarea al programador, ya que su visión de la memoria es similar a la de los sistemas uniprocesador: la comunicación es implícita vía cargas y almacenamientos, mientras que la sincronización se lleva a cabo de manera explícita a través de operaciones como cerrojos y barreras. Ahora bien, pese a que el código paralelo contiene información valiosa acerca de qué variables son las que se utilizan con fines de sincronización (cerrojos y barreras), los sistemas generalmente ignoran dicha información y tratan de manera uniforme todas las variables compartidas del programa.

El coste de implementación en hardware de mecanismos destinados a la sincronización eficiente ha supuesto hasta ahora una razón de peso para que los diseñadores de sistemas descarten dicha opción de las arquitecturas de memoria compartida. En aras de la simplicidad, la aproximación que vienen realizando es la de utilizar el soporte minimalista ofrecido por el microprocesador -en forma de instrucciones atómicas o primitivas de sincronización- para construir la sincronización en software. Este enfoque simplista hace que desde un punto de vista hardware, durante la ejecución de las aplicaciones paralelas no haya diferencia alguna entre las variables empleadas para sincronización y el resto de variables compartidas del programa; al fin y al cabo, todo son accesos a memoria que se tratan de manera uniforme, a pesar de las propiedades particulares de las variables de sincronización. Éstas tienen patrones de acceso y com-

partición diferentes a las variables normales, y puesto que a menudo están asociadas con la comunicación inter-procesador, los protocolos basados en invalidación no son óptimos [LLJ+92].

Las soluciones en software logran mantener la simplicidad en el hardware a costa de implementar las sincronización mediante algoritmos de listas o colas de peticionarios de candados, tal como se menciona en el capítulo 2. El problema de estos métodos de adquisición es la elevada latencia de adquisición del cerrojo en ausencia de contención, es decir, cuando en determinadas fases de la ejecución sólo hay un procesador adquiriendo y liberando el cerrojo repetidamente.

Como veremos en el capítulo 5, las aplicaciones paralelas tienen patrones de uso de cerrojos muy diferentes unas de otras: el número de cerrojos, la frecuencia de uso de los mismos, el número de procesadores que usa cada cerrojo, etc. son parámetros que varían ampliamente de unos programas a otros. También dentro de una misma aplicación diferentes cerrojos en el mismo instante tienen cargas distintas, o incluso el mismo cerrojo puede soportar más o menos contención en diferentes instantes de la ejecución de una misma aplicación. Así, elegir el algoritmo de sincronización más eficiente es una tarea difícil ya que su rendimiento depende de factores dinámicos como la contención o el tiempo de espera, muy difíciles de predecir. Por esta razón, decantarse por una solución software escalable pero con una elevada latencia de adquisición puede tener un efecto contraproducente en el rendimiento. Una propuesta que trata de solventar este problema es el uso de sincronización reactiva [LA94], mediante la selección dinámica del mecanismo más adecuado empleado, dependiendo de las circunstancias de ejecución en cada momento, con el fin de alcanzar un mayor rendimiento.

En este estudio nosotros evaluamos un mecanismo de adquisición simple como es test&test&set, que tiene un rendimiento óptimo en situaciones de baja contención, y abordamos el problema de escalabilidad mediante la colaboración por parte del controlador de directorio, con el fin de mantener el método de adquisición simple.

3.2. El directorio como árbitro de cerrojos

La idea de utilizar el directorio como guardia del tráfico de sincronización aprovecha las circunstancias que se dan durante la transferencia de un cerrojo test&test&set cuando hay numerosos peticionarios compitiendo por su adquisición. En este escenario de contención, prácticamente todas las peticiones de adquisición y liberación pasan por el directorio. Puesto que los peticionarios en espera ocupada tienen el bloque en estado compartido, sus intentos de adquisición (test&set) tras detectar la liberación (fase de test) han de conseguir la propiedad exclusiva del bloque, así que las peticiones al sistema de memoria necesitan llegar hasta el directorio. La liberación del cerrojo también provoca una escritura que falla en caché, pues probablemente el bloque fue invalidado poco después de la adquisición: el actual propietario obtuvo el cerrojo por-

que su test&set fue el primero en llegar al directorio en el turno anterior, y por tanto el siguiente test&set en llegar (el primero en ver el cerrojo ocupado en este turno) provocó la invalidación del bloque en la caché del nuevo propietario.

Supuesto que todos los accesos a cerrojos contendidos llegan hasta el directorio, éste tiene a su disposición la información necesaria para hacer el reparto de este recurso de una manera más inteligente, dando paso a uno y obligando a otros a esperar, evitando así el continuo envío de peticiones de búsqueda del bloque e invalidaciones. La única condición para realizar este encolado dinámico de los peticionarios es que el directorio sea capaz de distinguir los accesos a cerrojos del resto de accesos a variables convencionales del programa. Algunas alternativas para conseguir que los accesos a memoria debidos a actividades de sincronización sean distinguibles por el hardware se comentan en la subsección 3.2.2. En el siguiente punto describimos las ventajas que se obtienen como consecuencia de colocar el soporte para la sincronización eficiente a nivel del directorio, y criticamos varias limitaciones de este enfoque.

3.2.1. Soporte del directorio: Ventajas e inconvenientes

Tal como señalamos en el capítulo 2, existen diversas propuestas hardware para mejorar la eficiencia de las operaciones de sincronización mediante cerrojos. Varios estudios previos proponen la implementación de una cola de peticionarios del candado en caché [GVW89] [RL96] [KBG97], con la desventaja de la complejidad que introduce en el sistema de memoria dentro del microprocesador. Otras soluciones más recientes se basan en la especulación para afrontar el problema de la sobrecarga de sincronización [RG01] [MT01] [Raj02] [RS02]. En cualquier caso, todos estos mecanismos necesitan en mayor o menor medida la introducción de hardware específico al nivel del procesador y/o de las cachés, complicando su diseño y aumentando su consumo de energía.

Alternativamente, es posible alejar esta complejidad del núcleo del procesador y las cachés, y trasladarla a niveles inferiores de la jerarquía de memoria utilizando el directorio para almacenar la cola de peticionarios del cerrojo. Un esquema de este estilo ya fue implementado en el multiprocesador Stanford DASH [LLJ+92]. En esta máquina, los cerrojos usan la información apuntada en el directorio junto con operaciones de adquisición y liberación reconocibles por el hardware para mejorar el rendimiento, haciendo que la operación de liberación del cerrojo sólo cause invalidación en uno de los peticionarios.

La aproximación de llevar cuenta de los peticionarios desde el directorio introduce un nivel de indirección adicional a la hora de la transferencia del cerrojo, ya que siempre se tiene que pasar primero por el directorio. En comparación con los enfoques basados en caché, este enfoque tiene una latencia de adquisición mayor y un ancho de banda más reducido. En lo que respecta a las soluciones basadas en especulación, este enfoque difiere en que puede causar el llamado *efecto convoy* ya que requiere

siempre la adquisición del candado, y no puede explotar secciones críticas protegidas conservativamente.

Ahora bien, si implementar el soporte hardware a nivel del directorio supone de partida una ligera limitación del rendimiento, hacer que el mecanismo de optimización de la sincronización sea completamente independiente al núcleo acarrea otras ventajas tan importantes como las prestaciones:

- En primer lugar, desde el prisma del diseñador de un procesador de propósito general, la idea de introducir hardware especializado a nivel del núcleo para una tarea tan particular como la sincronización no es atractiva: la propia filosofía de diseño general es reacia a la inclusión de soporte especializado. Aunque el ámbito al que se destine el nuevo procesador sea el ser bloque de construcción de un sistema multiprocesador, no se justifica totalmente el elevado coste de implementación, ya que el gasto ha de verse amortizado por el número de chips colocados en el mercado. Dedicar una parte de los recursos del chip a tareas específicas puede apartar al microprocesador de otros sistemas que no necesitan de esta lógica extra. Por tanto, es beneficioso hacer que el soporte para la sincronización eficiente sea independiente al núcleo del procesador.
- Junto a esto, situar el soporte hardware para la sincronización al nivel del directorio tiene la ventaja de que se convierte en una solución válida para cualquier sistema paralelo que utilice directorios para mantener la coherencia de las cachés. No sólo los multiprocesadores de memoria compartida distribuida escalables (ccNUMAs) pueden beneficiarse de una propuesta de este estilo, sino que también sería aplicable a multiprocesadores en un único chip (CMPs, *chip multiprocessors*). Un mecanismo de sincronización eficiente que sea independiente del núcleo y la jerarquía de caché resulta especialmente atractivo en entornos de CMP, ya que uno los pilares básicos de estos sistemas es el diseño sencillo, minimalista de los núcleos de procesamiento integrados en el chip.

3.2.2. Sincronización explícita en hardware: Alternativas

Una de las formas que tiene el hardware de conocer cuándo se está accediendo a un cerrojo es extender la arquitectura del repertorio de instrucciones (ISA) para incluir un par de instrucciones cuya semántica sea *adquirir_cerrojo* y *liberar_cerrojo*, si bien una modificación en el ISA de este estilo es una pretensión bastante exigente sobre el diseño de cualquier arquitectura.

Otra posibilidad pasa por el uso de hardware específico para detectar cuando un procesador está realizando espera activa (*spinning*), como describe una reciente propuesta [LLS06]. Este mecanismo es capaz de detectar en qué momento se está realizando la sincronización, y se podría particularizar para averiguar sobre qué direcciones se lleva a cabo el *spinning* e informar al controlador de caché sobre qué líneas

contienen variables candado, de manera que fuese posible marcar de alguna forma los mensajes enviados al sistema de memoria para notificar al directorio de este hecho.

Por otro lado, si no es posible contar con la colaboración del procesador para conocer exactamente qué direcciones de memoria contienen cerrojos, se podría aplicar la técnica de predicción o inferencia para pronosticar qué peticiones se refieren a candados, utilizando el propio valor de las variables y el patrón de acceso a las mismas para efectuar la predicción. Con el apoyo del software o del compilador podría conseguirse una alta tasa de acierto en la predicción, mediante el uso de *firma* característica que acompañe a cada cerrojo, a modo de envoltura. En cualquier caso, el hardware colocado en el directorio tendría que contar irremediablemente con mecanismos de respaldo (como temporizadores) para recuperarse ante predicciones erróneas y restaurar la política de gestión de memoria convencional, garantizando tanto la corrección del programa como la ausencia de interbloqueos. Un trabajo que utiliza la predicción como técnica para mejorar la eficiencia de la sincronización es [RKG04], donde se proponen los *cerrojos encolados inferencialmente* (IQL), aunque en dicho trabajo el hardware para retrasar la respuesta a las peticiones de adquisición se coloca a nivel de controlador de caché.

Finalmente, por citar una caso real, el Stanford DASH [LLJ+92] es capaz de distinguir los accesos a cerrojos a nivel del directorio gracias a que los cerrojos residen en un espacio de memoria separado del resto de variables comunes. No obstante, la razón primordial de esta decisión de diseño es que su procesador (el MIPS R3000) no soporta ninguna operación de lectura-modificación-escritura atómica.

Partiendo de la premisa de que el controlador de directorio es capaz de distinguir las operaciones de adquisición y liberación sin ambigüedad alguna, en la siguiente sección describimos el funcionamiento de este *controlador de cerrojos* acoplado al controlador de directorio. En el capítulo 5 evaluaremos el rendimiento tope que se puede alcanzar utilizando este *director de orquesta* en la transferencia de cerrojos.

3.3. El controlador de cerrojos

En este apartado describimos el protocolo que hemos diseñado para llevar a cabo una transferencia del cerrojo más efectiva en situaciones de alta contención. El método de adquisición utilizado en todo caso es test&test&set, de forma que en ausencia de contención su funcionamiento no se perturba y se consigue una latencia mínima. El mecanismo que describimos a continuación entra en acción cuando se estima que el número de peticionarios del cerrojo supera un cierto umbral. La idea básica es modificar la política de gestión de las líneas de memoria que realiza el directorio, cuando dichas peticiones se refieren a operaciones de sincronización. Se trata de aplicar una política de gestión de la memoria diferente sobre los bloques que contienen variables cerrojos, cuando se detecta que hay contención en el acceso a los mismos.

En una arquitectura basada en directorio convencional, como por ejemplo el SGI Origin 2000, el controlador de directorio aplica una política de gestión de líneas de memoria uniforme, la misma para todas las peticiones que le llegan, ya que para el directorio no hay diferencia alguna entre bloques. La arquitectura que describimos en esta sección, por contra, no utiliza la misma política para todas las líneas, sino que distingue entre las peticiones debidas a operaciones de sincronización de cerrojo, y el resto. Según el protocolo que proponemos, las líneas de memoria que contienen variables cerrojo (en adelante, *líneas cerrojo*) se manejan de manera diferente al resto de bloques, debido a las propiedades particulares de este tipo de variables, como es el patrón de acceso por parte de los procesadores.

En cualquier caso, asumimos que cada línea cerrojo contiene una sola variable y no alberga ningún otro dato, es decir, existe un relleno alrededor de cada variable cerrojo de al menos el tamaño del bloque de memoria. Este punto es fácil de conseguir modificando el código de la macro que contiene la declaración del cerrojo (`LOCKDEC`) para añadir el relleno. De esta forma se evitan problemas de falsa compartición entre otras variables del programa y el cerrojo, o entre varios cerrojos, y se simplifica el diseño del protocolo cuyo funcionamiento describimos más abajo. Esta asunción es similar al enfoque adoptado en el diseño del DASH [LLJ+92], donde en el espacio de memoria separada destinado a cerrojos sólo se proporciona un cerrojo por bloque de memoria. En cuanto a las consecuencias de utilizar relleno en ciertos bloques de memoria, cabe decir que por lo general una aplicación accede en cada instante a un reducido número de cerrojos y por tanto la infrautilización de parte de la memoria caché no representa un problema significativo.

La lógica acoplada al controlador de directorio para modificar la política de gestión de cada petición de memoria la hemos denominado *controlador de cerrojos*. Así, mientras el controlador de directorio actúa de manera convencional sobre todas las peticiones, el controlador de cerrojos las fisgonea para reconocer aquellas que se refieren a líneas cerrojo y llevar cuenta del número de procesadores que están accediendo simultáneamente a cada cerrojo en uso. Así, cuando se detecta que la contención supera un cierto umbral establecido (más de k procesadores intentando adquirir el cerrojo a la vez), el controlador de cerrojos informa al controlador de directorio para que cambie la política de gestión de esa línea de memoria y la adecúe a las propiedades particulares de las variables cerrojo.

Este tratamiento especial de las líneas o *política de gestión de bloques cerrojo* se basa en retrasar la respuesta por parte del controlador de directorio a las peticiones de adquisición recibidas, y responder cada vez a un nodo, otorgando siempre un cerrojo libre. En otras palabras, el controlador de directorio *encola* las peticiones de adquisición sobre cerrojos contendidos y las responde una a una conforme el cerrojo es liberado por el último propietario. Así se consigue que el hardware sea capaz de convertir dinámicamente cerrojos test&test&set en cerrojos basados en cola, y resolver en gran parte el problema de escalabilidad que este sencillo método de adquisición presenta cuando la contención es alta.

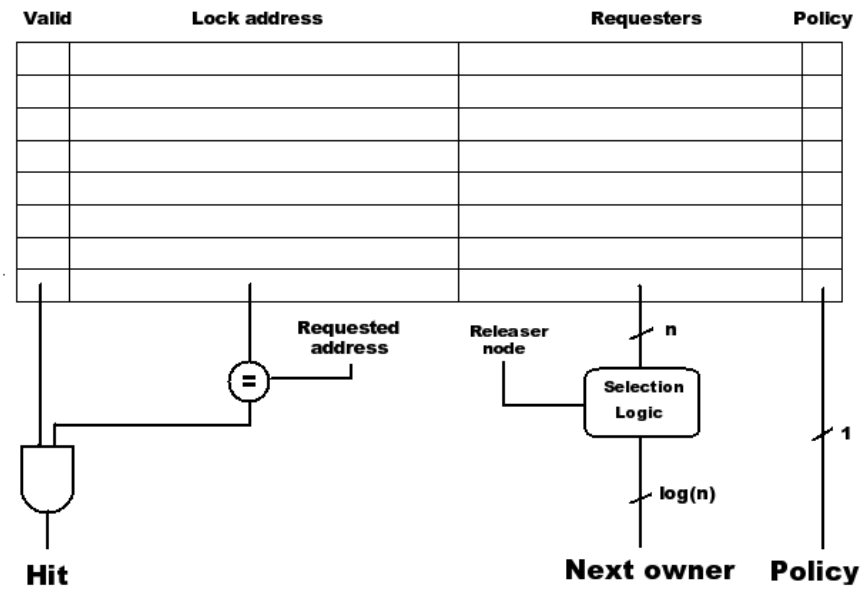

Figura 3.2: Caché del controlador de cerrojos.

3.3.1. Funcionamiento

El controlador de cerrojos se puede entender como una unidad hardware acoplada al controlador de directorio, capaz de observar los mensajes que este último envía y recibe. Internamente el controlador de cerrojos contiene una pequeña memoria en la que guarda las direcciones de los cerrojos más recientemente usados, junto con cierta información sobre cada cerrojo. Esta información consiste en un bit que indica la política de gestión de memoria que se está aplicando sobre ese cerrojo actualmente, junto con un vector de n bits, uno por cada nodo del sistema. La semántica de este vector cambia según la política en uso: bajo manejo regular, el vector indica qué nodos han solicitado la adquisición del cerrojo desde que se recibió la última liberación en el directorio; si se aplica política de cerrojos, el vector registra aquellos nodos que enviaron sus peticiones de adquisición y están a la espera de obtener la respuesta que les otorgue el candado. La estructura detallada de la memoria interna al controlador de cerrojos se aprecia en la figura 3.2.

Ante la recepción de una petición referente a una línea cerrojo, y en paralelo con el funcionamiento normal del controlador de directorio, el controlador de cerrojos accede a su memoria interna para comprobar si tiene algún tipo de información acerca de este cerrojo, y en función de esta información, establece una de sus salidas: se trata de un bit que controla la política que debe aplicarse sobre el bloque de memoria en cuestión. Para todas las peticiones que no acceden a bloques cerrojo, la salida será siempre 0 y simplemente se dejará al controlador de directorio actuar como de costumbre. En cambio, cuando la petición entrante es un acceso a una línea que contiene un cerrojo, ya sea una adquisición (RMW), liberación (WRITE) o espera ocupada (READ), si la salida del controlador de cerrojos es positiva variará la forma en que se responde

a este mensaje. El autómata de estados que describe el protocolo utilizado sobre las líneas cerrojo se muestra en la figura 3.3.

Veamos lo que ocurre ante la recepción de cada uno de los tres tipos de petición referidos a cerrojos. En los tres casos, el controlador de cerrojos accede a su memoria interna para comprobar si tiene información acerca de este cerrojo.

Fallo en la caché del controlador de cerrojos

ADQUISICIÓN. Se asume que o bien se trata del primer acceso al cerrojo, o bien que éste no se ha usado durante algún tiempo y su entrada ha sido reemplazada por otro cerrojo. En cualquier caso, la política de gestión de esa línea de memoria será convencional. Se crea una nueva entrada con la dirección de este cerrojo, se establece el vector de peticionarios vacío, y se desactiva el bit que indica si se aplica política tipo-cerrojo sobre esa línea de memoria. Si no hay ninguna entrada libre, se elige la entrada menos recientemente usada de entre aquellas que no tienen la política de cerrojo activa como víctima del reemplazo, sin que sea necesario realizar ninguna acción al respecto. Si todas las entradas están siendo usadas bajo la política de cerrojos, no hay reemplazo: sencillamente este nuevo cerrojo será tratado con la política convencional.

LIBERACIÓN/ESPERA ACTIVA. Ambos tipos de peticiones son manejados por el controlador de directorio, como si se tratase de accesos a cualquier otra variable común del programa.

Acierto en la caché del controlador de cerrojos

ADQUISICIÓN. En este caso las acciones a tomar dependen en primera instancia de la política de manejo del cerrojo en ese instante, así que lo primero que se consulta es el bit que indica la política aplicada.

- **Política de cerrojos inactiva.** El controlador de cerrojos no interviene en la respuesta a la adquisición, deja que sea el controlador de directorio el que se encargue de este mensaje.
- **Política de cerrojos activa.** Aquí entra en funcionamiento el denominado *encolado dinámico de peticionarios del cerrojo*. En primer lugar se observa si el cerrojo está actualmente libre u ocupado.

  Si hay algún bit activo en el vector de peticionarios, significa que el cerrojo está en propiedad de alguno de esos nodos, así que el nuevo peticionario se añade a la cola, activando el bit correspondiente. El controlador de cerrojos inhibe el funcionamiento normal del directorio y provoca que no se envíe respuesta alguna

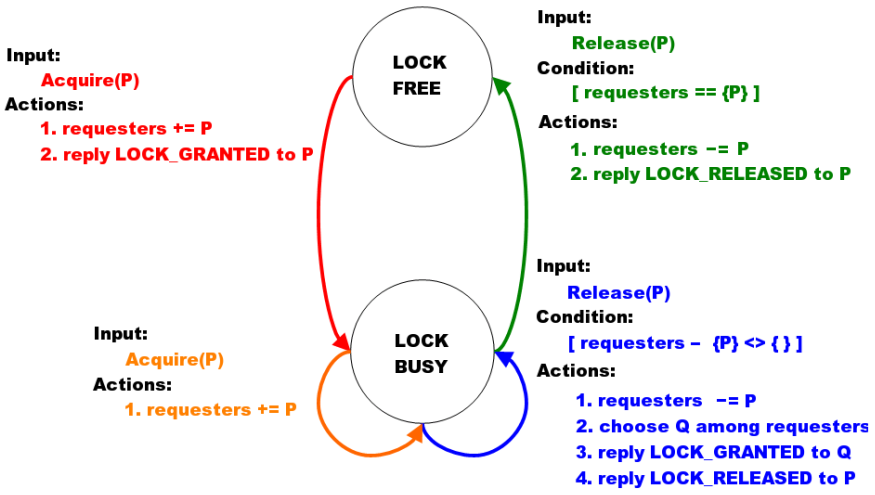

Figura 3.3: Autómata para una línea con política de cerrojo.

al nuevo peticionario, con lo que éste queda bloqueado en espera de satisfacer el acceso a memoria RMW. Por motivos de sencillez y claridad, la lista de nodos a la espera del cerrojo descrita aquí se implementa mediante el *vector de peticionarios*, aunque bien se podría utilizar la propia entrada de directorio del bloque con este fin.

Si la información de directorio indica que ninguna caché tiene copia del bloque, el cerrojo está libre y por tanto el controlador de cerrojos responde al peticionario inmediatamente. Se activa el bit del vector de peticionarios. El mensaje de respuesta, denominado `LOCK_GRANTED`, no contiene datos sino que en su lugar indica al controlador de caché que genere automáticamente una línea de memoria con un valor de cerrojo libre. Este mensaje provoca a su recepción no solo la generación del bloque de datos *virtual*, sino también que el controlador de caché invalide la copia del bloque inmediatamente después de satisfacer el acceso a memoria causado por la adquisición del cerrojo (instrucción RMW). Así, la eventual liberación (escritura) del cerrojo fallará en caché y requerirá el envío de un mensaje al directorio, y así éste podrá dar paso a la sección crítica al siguiente nodo de la cola de espera.
Al utilizar un mensaje de respuesta sin datos, se elimina la necesidad de acceder a memoria durante la transferencia del cerrojo y se consigue que la solución sea independiente de la latencia de la misma. Es más, si el controlador de directorio (y por tanto el de cerrojos) se integran en el mismo chip del procesador, como ocurre en la arquitectura que nosotros proponemos, la transferencia del cerrojo estará limitada fundamentalmente por el retardo introducido por la red de interconexión, es decir, por la distancia entre el nodo *home* del bloque cerrojo y los nodos origen y destino de la transferencia.

LIBERACIÓN. El funcionamiento del controlador de directorio ante la recepción de un mensaje de liberación también depende de la política con que se esté manejando la línea de memoria cerrojo en ese instante.

- **Política de cerrojos inactiva.** Las peticiones de liberación son utilizadas por el controlador de cerrojos como indicadores del nivel de contención del cerrojo. Esto se basa en una circunstancia que se da fundamentalmente en los cerrojos poco contendidos, y es que la liberación del cerrojo encuentra el bloque de caché en propiedad, a menos que éste haya sido víctima de un reemplazo durante la ejecución de la sección crítica. Esto a su vez es bastante improbable, ya que las secciones críticas suelen ser cortas y la variable cerrojo es una de las más recientemente usadas en el programa. Así, si la adquisición del cerrojo obtuvo una copia del bloque en propiedad y la liberación no lo encuentra en ese mismo estado, casi con seguridad se debió a que otro procesador intentó adquirir el cerrojo mientras el primero se encontraba dentro de la sección crítica[1]. Debido al mensaje de coherencia generado por el intento de adquisición fallido, el bloque es invalidado en la caché del poseedor del cerrojo antes de que la liberación se lleve a cabo, y esto hace que la escritura (liberación) falle en caché y se envíe un mensaje al directorio.
  En este razonamiento se basa la detección de cerrojos contenidos y el cambio de la política aplicada sobre éstos. Cuando llega una liberación al directorio sobre un cerrojo al que se le está aplicando la política convencional, el controlador de caché activa el bit correspondiente al nodo liberador en el vector de bits que anteriormente denominamos *de peticionarios*, pero que en este caso tiene una semántica diferente. Los bits activos no permanecen así indefinidamente sino que mientras la política no cambia, cada vez que llega al directorio una adquisición, se borra el bit asociado al nodo que realiza la petición de adquisición. Por tanto, ante la llegada de una liberación en esta situación, el controlador de cerrojos chequea cuántos bits del vector de *liberaciones en directorio* están activos. A partir de un cierto número de bits k, parámetro que denominamos *umbral de contención*, se activa el bit que señala la aplicación de la política tipo cerrojo para esa línea, y se borran todos los bits del vector de peticionarios. Si bien queda claro que k será un número pequeño, es necesario llevar a cabo una evaluación para decidir a partir de qué grado de contención el rendimiento de test&test&set es peor que el del método de encolado dinámico a nivel de directorio.

- **Política de cerrojos activa.** En primer lugar se desactiva el bit del vector de peticionarios correspondiente al nodo liberador. A continuación, se observa el vector de peticionarios para ver si hay otros nodos esperando para adquirir el candado. Si efectivamente queda algún bit activo en dicho vector, se selecciona uno de los candidatos como nuevo propietario del cerrojo y se le envía un mensaje `LOCK_GRANTED`. Como se comentó anteriormente, este mensaje le permite

[1]Estamos asumiendo siempre que cada cerrojo se encuentra en un bloque rodeado de relleno y por tanto no entra en falsa compartición con ninguna otra variable.

resolver el acceso a memoria RMW que paralizó el procesador instantes antes y a través del cual trató de adquirir el cerrojo, y por tanto le otorga acceso a la sección crítica. No obstante, merced a la orden de invalidación instantánea del bloque tras la satisfacción del acceso a memoria, el nuevo propietario queda obligado a notificar al directorio del momento en el que sale de la zona de exclusión mutua, dado que la escritura de liberación no encontrará el bloque en su caché.
Paralelamente al envío de `LOCK_GRANTED` al nuevo poseedor, el controlador de cerrojos también envía un mensaje al anterior propietario, en respuesta al fallo de escritura (liberación). Este mensaje, que hemos nombrado `LOCK_RELEASED`, es similar a `LOCK_GRANTED`: indica al controlador de caché que ha de generar un bloque de datos *virtual* en estado exclusivo, completar la escritura (liberación) e invalidar a continuación el bloque.

ESPERA ACTIVA. Si la política activa es la estándar, el controlador de directorio responde como lo hace ante cualquier otro acceso de lectura. En cambio, si el bloque está siendo gestionado según la política de cerrojos, entonces nunca se recibirá un mensaje de este estilo en el directorio. Es posible no obstante que el procesador emita al sistema de memoria una lectura especulativa (test) del valor del cerrojo, antes de resolver el acceso de escritura (test&set) con que se inicia la llamada `LOCK()`. En cualquier caso, puesto que ambos accesos a memoria se refieren al mismo bloque, el fallo de lectura siempre se mezcla en el MSHR (*Missing Status Handling Register*) utilizado para manejar el fallo de escritura precedente, y por tanto no se envía ninguna petición de lectura al directorio.

3.3.2. Selección del siguiente propietario

Cuando un bloque cerrojo está siendo manejado con la política propuesta, al recibir un mensaje de liberación procedente de un nodo P, el controlador de cerrojos borra el bit P-ésimo del vector de bits usado como *lista de peticionarios* y examina en orden ascendente los bits desde de la posición P+1 hasta P-1, volviendo al bit 0 al llegar al final de la ristra. Si en este recorrido encuentra un bit activo en la posición Q, se entiende que ese nodo solicitó la adquisición del cerrojo y se encuentra en espera, así que se elige como siguiente propietario y se le envía un mensaje de concesión para otorgarle acceso a la sección crítica.

Aunque esta política de selección del nuevo propietario no sigue un esquema FIFO, sí que puede considerarse justa puesto que da exactamente la misma oportunidad de adquisición del cerrojo a todos los procesadores. Así, si un nodo P solicita el cerrojo habiendo un nodo Q a la espera, en el instante en que un tercer nodo N es el propietario, siendo $id(N) < id(P) < id(Q)$, P adquirirá el cerrojo antes que Q. No obstante, puesto que los procesadores van entrando a la sección crítica en orden sucesivo de identificador a partir del *id* del que libera, el mecanismo puede

considerarse justo, y garantiza que no hay inanición. Además, por la forma secuencial en que se selecciona el siguiente nodo, esta política es muy sencilla de implementar en hardware.

3.3.3. Detección del cese de la contención

Los cerrojos atraviesan estados de contención diferentes a lo largo de la ejecución de una aplicación. En este sentido, el mecanismo dinámico de cambio de la política de gestión de las líneas de memoria debe adaptarse a los cerrojos contendidos, pero también debería ser capaz de retomar su comportamiento habitual en épocas en las que no hay muchos procesadores compitiendo por el mismo cerrojo. Por ejemplo, hay aplicaciones paralelas cuyos hilos acceden de forma irregular a los datos, y otras en las que las operaciones de sincronización están colocadas conservativamente, protegiendo excesivamente los datos compartidos con cerrojos. En estos programas, puede darse el caso de que un mismo procesador necesite acceder repetidamente a la misma sección crítica, sin que ningún otro procesador del sistema compita con él. En este caso, si el cerrojo ha sido previamente encolado debido a un periodo de contención, el tiempo de adquisición de la fase no contendida se verá claramente penalizado con respecto a lo que se obtendría con la política normal: al estar activa la política de encolado sobre esta línea, cada adquisición y liberación tendrá que pasar por el directorio, mientras que con el funcionamiento normal el procesador obtendría una copia en propiedad y las autotransferencias del cerrojo se resolverían localmente.

El mecanismo de controlador de cerrojos, tal y como se ha descrito hasta el momento, no soporta el regreso a la política estándar una vez que se ha activado la estrategia de encolado dinámico sobre ella. Sin embargo, es bastante sencillo conseguir que tenga un comportamiento reversible añadiendo muy poco hardware extra. Se trata de guardar por cada cerrojo el identificador de su último propietario y utilizar un contador para conocer cuántas veces se ha producido la auto-transferencia del cerrojo, es decir, cuántas veces el cerrojo ha sido liberado y vuelto a adquirir por el mismo procesador.

Para conseguir esto, cada vez que recibe una petición de liberación el controlador de cerrojos anota en un campo llamado *último_propietario* el identificador del nodo liberador. Si el valor de este campo coincide con el que se va a anotar, significa que el mismo nodo ha sido propietario del cerrojo dos veces consecutivas, así que se incrementa el contador de auto-transferencias del cerrojo. Si el valor almacenado en *último_propietario* no coincide con el identificador del nodo liberador actual, se actualiza el campo con el nuevo *id* y el contador de autotransferencias se pone 0. Cuando el contador de auto-transferencias sobrepasa un cierto valor w, la política aplicada sobre esa línea vuelve a ser la convencional, y la respuesta a la liberación es un mensaje normal, que otorga al nodo una copia en propiedad del bloque sin la obligación de invalidarla.

3.3.4. Bloqueo de los peticionarios: Alternativas

Bajo política de transferencia eficiente de líneas cerrojo, el controlador de cerrojos retrasa la respuesta a las nuevas peticiones de adquisición que le llegan si el candado solicitado está ocupado. No es hasta que la lógica de selección del siguiente propietario estima oportuno conceder acceso a un peticionario, que se envía respuesta a dicha petición de adquisición ignorada temporalmente. Por tanto, durante todo el tiempo que el nodo está en cola, la instrucción RMW no resuelve su acceso a memoria y no es capaz de completar su ejecución. Pese a que el procesador intenta ocultar la latencia de este acceso obteniendo y ejecutando especulativamente otras instrucciones, eventualmente el buffer de reordenación (ROB) se llena y el procesador se detiene.

Otra alternativa de diseño del protocolo es hacer uso de *incoherencia* y *actualización* para bloquear a los peticionarios y liberarlos después. En lugar de retrasar el envío de la respuesta al RMW hasta que se decide otorgar acceso a la sección crítica, podríamos haber optado por utilizar un mensaje `LOCK_DENIED` que contenga un bloque de datos auto-generado, con un valor del cerrojo ocupado. Así, cada peticionario en espera contaría con una copia del bloque sobre la cual realizar la espera activa localmente, en su propia caché. Esto lleva a la presencia de varias copias del mismo bloque cerrojo con diferentes valores, es decir, copias *incoherentes* del bloque. La utilización de incoherencia en un sistema NUMA con coherencia de cachés se basa en una de las propiedades particulares de las variables cerrojo: no siguen el comportamiento observado para las variables de datos del programa, sino que su semántica es binaria. Aprovechando esto, se puede relajar el protocolo de coherencia sobre dichas líneas de memoria cerrojo para hacer más efectiva la transferencia.

Por otro lado, en la propuesta de controlador de cerrojos que bloquea a los peticionarios del cerrojo retrasando la respuesta, asumimos en todo momento un protocolo de coherencia de caché basado en invalidación. En cambio, decantarse por la opción de incoherencia necesita introducir mensajes de actualización para aquellas líneas sobre las que no se aplica coherencia. Así, la transferencia del candado a otro procesador necesita que se actualice el valor del bloque en estado compartido sobre el que está realizando la espera activa.

En nuestra propuesta del controlador de cerrojos hemos optado por el retraso de las respuestas en lugar de hacer uso de incoherencia por dos razones fundamentales:

- Primero, retrasar la respuesta provoca que eventualmente el procesador se detenga, lo cual se traduce en un menor consumo de energía por parte de aquellos peticionarios del cerrojo que están en cola a la espera de acceder a la sección crítica. Esto no ocurre si utilizamos incoherencia, ya que en tal caso los nodos a la espera siguen ejecutando continuamente las instrucciones de la fase de test del método de adquisición: carga, comprobación y salto.
- En segundo lugar, la opción de usar incoherencia en un sistema con coherencia de cachés acarrea otra combinación nada habitual: el protocolo de coherencia

de caché es basado en invalidación, pero sobre las líneas no coherentes se aplica una política de actualización. Esto hace el diseño del controlador de caché más complejo, al tener que implementar y reaccionar de maneras muy diferentes ante líneas coherentes y líneas incoherentes.

3.3.5. Implementación de la lista de peticionarios

En el protocolo descrito anteriormente mencionamos la presencia de un vector de peticionarios del cerrojo, a partir de la cual el controlador de directorio elige al siguiente nodo para concederle acceso. En las secciones anteriores se ha hablado en todo momento de este vector de bits como parte integrante de cada entrada en la caché del controlador de cerrojos. La decisión de utilizar un vector de peticionarios por cada cerrojo, en lugar de la propia entrada de directorio, tiene su explicación en la sencillez pero también en la velocidad de acceso: el controlador de cerrojos, incluida su memoria interna, está implementado junto al controlador de directorio dentro del chip del procesador, según los parámetros de la arquitectura simulada. Así, puesto que el tiempo de acceso a las listas de peticionarios es reducido, la transferencia del cerrojo no sufre la latencia de acceso a la memoria *off-chip*, ya que no necesita ni leer el bloque de memoria ni obtener la información de directorio.

Sin embargo, supuesto que el directorio utiliza un vector de bits (*full bit vector*) como código de compartición, se puede sacrificar parte del rendimiento conseguido con la configuración anterior, al tiempo que se reduce la sobrecarga de hardware introducida: el vector de peticionarios se puede implementar sin hardware adicional haciendo uso de la propia entrada de directorio del bloque. El código de compartición *full-vector* consta de un bit por nodo en el sistema, que indica si dicho nodo tiene copia del bloque en un momento dado. La semántica del código de compartición para el caso de un bloque cerrojo es muy similar al resto de líneas de memoria: cuando se trata de un cerrojo, en lugar de anotar qué nodos tienen copia del bloque, anotamos qué nodos están a la espera de obtener el bloque, es decir, qué nodos son peticionarios del cerrojo. En el caso de que utilicemos la propia entrada de directorio para construir la cola de peticionarios, la recepción de un mensaje de adquisición del cerrojo procedente del nodo P siempre lleva asociada la activación del bit P-ésimo en la entrada de directorio, independientemente de la respuesta que el controlador de directorio envíe al nodo (mensaje de concesión o mensaje de denegación).

Capítulo 4

Entorno de evaluación

En este capítulo describimos la metodología y el entorno de evaluación que hemos utilizado para desarrollar nuestro trabajo. En particular, este apartado describe el simulador RSIM y las aplicaciones paralelas que hemos ejecutado sobre dicho simulador para realizar este estudio. Finalmente, comentamos las modificaciones que hemos llevado a cabo sobre el código fuente del simulador para implementar los mecanismos de sincronización eficiente descritos en el capítulo anterior.

Para evaluar el rendimiento de nuestra propuesta hemos empleado aplicaciones paralelas de memoria compartida escritas en C, extendidas con las macros PARMACS [ANMB97] y soportadas directamente por RSIM gracias al código de la librería de aplicaciones que se distribuye con el simulador. Estas aplicaciones son ampliamente usadas en la evaluación de arquitecturas multiprocesador de memoria compartida.

4.1. El simulador RSIM

El simulador RSIM [PRA97, HPRA02] (*Rice Simulator for ILP Multiprocessors*) fue desarrollado por la Universidad de Rice para estudiar tanto uniprocesadores que explotan agresivamente el paralelismo a nivel de instrucción como multiprocesadores de memoria compartida cc-NUMA. RSIM es capaz de ejecutar aplicaciones compiladas para la arquitectura SPARC V9/Solaris usando compiladores y enlazadores ordinarios SPARC, aunque con algunas excepciones. Inicialmente RSIM fue concebido para su uso en plataformas Sun Ultra-I/Solaris, HP PA-8000/HP-UX y MIPS R8000/IRIX. Posteriormente, el simulador fue portado al entorno x86/Linux [FG05] y es a partir de esta versión sobre la que hemos desarrollado nuestro estudio.

RSIM modela un procesador cuya microarquitectura explota agresivamente el ILP, que guarda especial parecido con el MIPS R10000. En el ámbito multiprocesador, RSIM simula un sistema de memoria compartida distribuida con coherencia de ca-

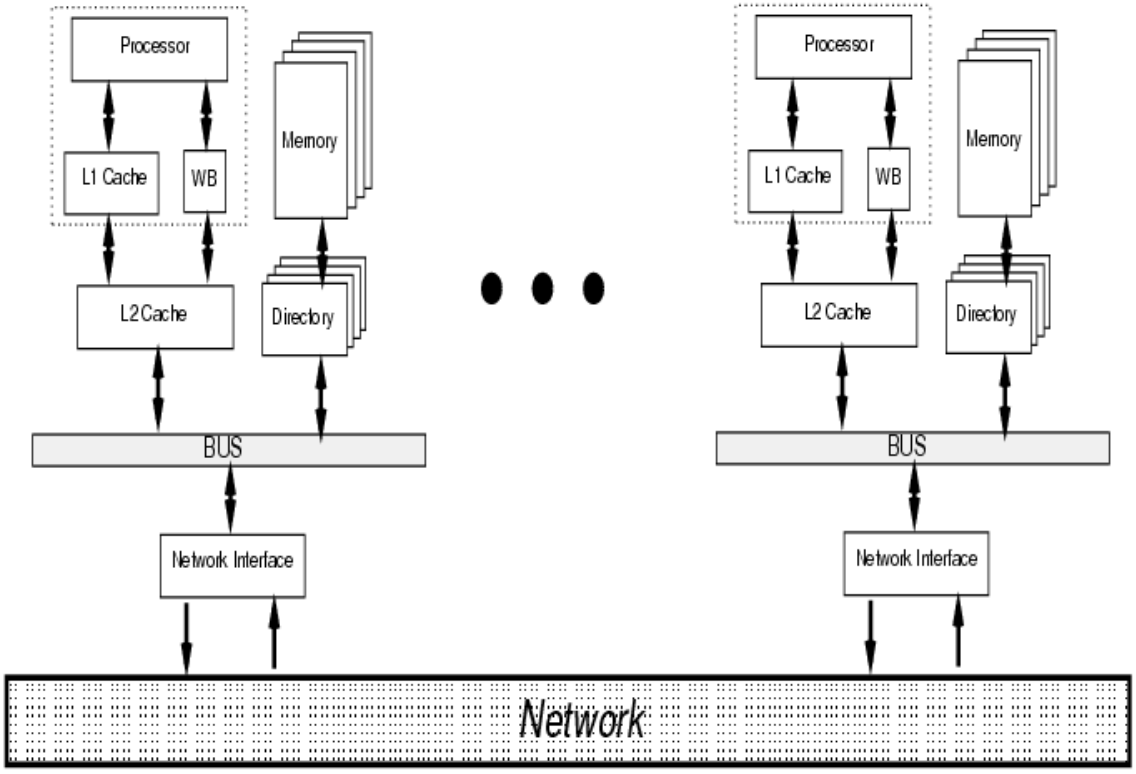

Figura 4.1: Arquitectura multiprocesador cc-NUMA modelada en RSIM.

ché en hardware (cc-NUMA). Cada nodo de procesamiento de la arquitectura RSIM consiste de un procesador, una jerarquía de memoria caché de dos niveles (con un buffer de escritura si la L1 es de escritura directa), una porción de la memoria física del sistema con sus entradas de directorio asociadas, y una interfaz de red. La caché L2, la memoria y directorio, y la interfaz de red están conectadas mediante un bus sobre el que tienen lugar las comunicaciones locales dentro del nodo. La interfaz de red conecta el nodo con una red de interconexión para las comunicaciones remotas. La arquitectura descrita se puede ver en la figura 4.1.

4.1.1. Microarquitectura del procesador en RSIM

RSIM modela una microarquitectura de procesador que trata de extraer de manera muy agresiva el paralelismo a nivel de instrucción en el flujo del programa. El cauce está segmentado en seis fases: *Fetch*, *Decode*, *Issue*, *Execute*, *Complete* y *Graduate*. Las principales características del procesador simulado son ejecución superescalar, planificación fuera de orden, renombramiento de registros, predicción de saltos estática y dinámica, y ejecución de cargas especulativa.

La mayoría de parámetros del procesador son configurables por el usuario: número de instrucciones emitidas en cada ciclo, número de unidades funcionales y sus

latencias, el tamaño de la lista de instrucciones activas, etc.

4.1.2. Jerarquía de memoria

Se trata de una jerarquía de dos niveles de caché en la que la L1 puede ser tanto de escritura directa sin búsqueda de bloque ((*write through, write no allocate*) como de post escritura con política de búsqueda de bloque (*write back, write allocate*). Las cachés son segmentadas y pueden ser configuradas por el usuario indicando su tamaño, el tamaño de la línea, la asociatividad, la latencia y el número de puertos. Es posible tener múltiples peticiones de acceso a memoria pendientes gracias a que su estado se almacena en unos registros de manejo del estado de los fallos (MSHR, *Miss Status Handling Register*), cuyo número es igualmente configurable. El acceso a la memoria y a la información de directorio se realiza en paralelo. La memoria principal también puede ser configurada indicando su tiempo de acceso, factor de intercalado, tiempo de acceso mínimo al directorio y tiempo que tarda el directorio en crear los distintos mensajes de coherencia. El protocolo de coherencia a nivel de directorio es MESI, similar al del SGI Origin 2000 [CS99], y permite transferencias caché a caché. Como código de compartición para el directorio se usa un vector con un bit por nodo del sistema *full-map*. Finalmente, RSIM soporta tres modelos de consistencia de memoria, consistencia secuencial, consistencia del procesador y consistencia relajada.

4.1.3. Red de interconexión

Para la comunicación entre los distintos nodos de la máquina, RSIM utiliza una malla de dos dimensiones con enrutamiento *wormhole* y conectada con varios conmutadores o *switches*. Este tipo de enrutamiento hace que la latencia de la red sea independiente de la distancia entre el nodo que envía y el que recibe, por lo menos en mensajes largos. Además, otra ventaja de este sistema de enrutamiento es que sólo requiere buffers muy pequeños en los conmutadores. La red de interconexión permite la conficuración de parámetros como su ancho de banda, el tamaño de los buffers de cada conmutador y la longitud de la cabecera de los paquetes de control.

4.1.4. Módulos en RSIM

Cada módulo en RSIM actúa de forma independiente. Todos los módulos tienen una serie de puertos que usan para comunicarse con los otros módulos mediante el envío de mensajes de petición y respuesta. Los puertos actúan como unas colas que contienen mensajes. Si un módulo está comunicado con otro compartirán el mismo puerto y uno de ellos insertará mensajes en la cola (puerto de entrada) y otro de ellos los irá eliminando de la cola (puerto de salida).

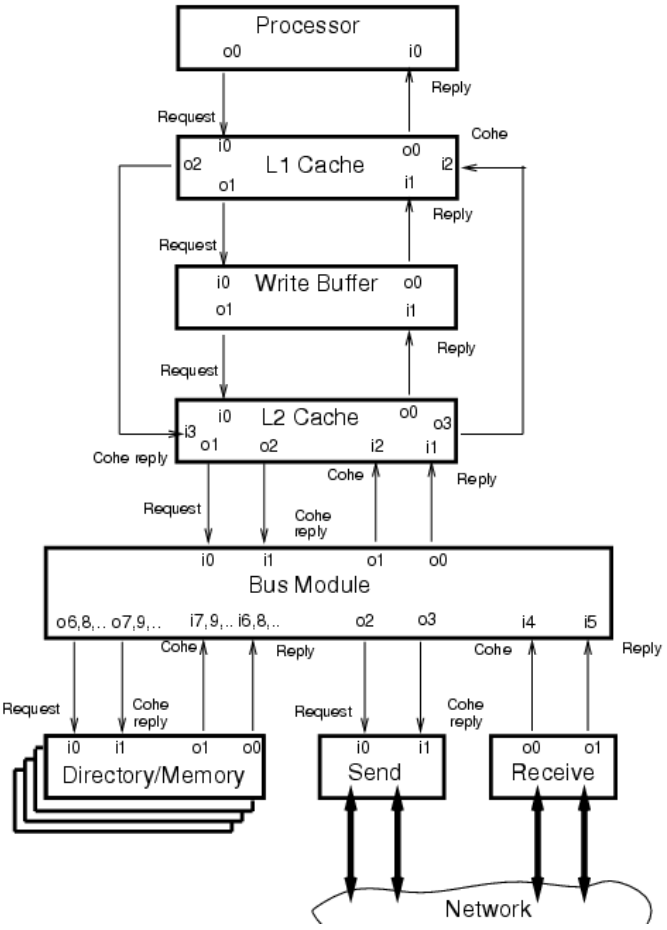

Figura 4.2: Módulos y conexiones de puertos en RSIM.

En la figura 4.2 se muestran los diferentes módulos de RSIM así como los puertos por defecto que los interconectan. Si un puerto es usado como salida su nombre comienza por "o" y si es usado como entrada empieza por "i".

4.2. Modificaciones realizadas

Para evaluar el mecanismo de encolado dinámico de cerrojos, así como para caracterizar de forma detallada el rendimiento de este tipo de sincronización en las aplicaciones paralelas consideradas, ha sido necesario modificar el código fuente del simulador. El tamaño y la complejidad de este simulador hacen que cualquier cambio o extensión en su código requiera estar muy bien familiarizado con varios ficheros y múltiples funciones del código, lo cual requiere un tiempo y esfuerzo considerable. Nos ha sido necesario aprender de forma precisa el funcionamiento de buena parte del simulador antes de atrevernos si quiera a realizar cambios. Por esta razón, el tiempo de desarrollo de la versión con soporte para encolado dinámico de cerrojos ha sido largo,

y el proceso de depuración laborioso. Nos hemos familiarizado prácticamente con toda la parte del simulador que concierte al sistema de memoria, incluyendo la anatomía de los mensajes, el funcionamiento de ambos niveles de caché y el controlador de directorio.

Junto a esto, el simulador nos ha impuesto otras limitaciones que han dificultado la evaluación y alargado el proceso de desarrollo: nos referimos al tema de la recompilación de los *benchmarks*. En este estudio hemos tenido que trabajar en todo momento con el mismo conjunto de programas binarios compilados específicamente para RSIM. Esto ha supuesto un problema ya que, con el fin de evaluar la influencia de la falsa compartición en el rendimiento de los cerrojos, necesitamos colocar cada cerrojo en un bloque de memoria diferente. Sin embargo, no fue posible hacer estos cambios desde el código fuente de las aplicaciones, o desde la propia definición de las macros, pues cualquiera de estos cambios requiere recompilar la aplicación. El problema es que los binarios resultantes sencillamente no funcionaban en el simulador. Esto es debido entre otras cosas a las diferencias entre las versiones del repertorio SPARC soportado por RSIM (1997) y el ISA que utilizan los compiladores de que disponemos, mucho más actuales.

Nuestro trabajo de programación sobre el simulador RSIM se puede dividir en tres partes:

- El código para la obtención de estadísticas precisas sobre el uso de cerrojos, que nos ha permitido realizar una precisa caracterización de este tipo de sincronización en las aplicaciones paralelas, mostrada en el capítulo 5.
- El código que implementa la funcionalidad de encolado dinámico de cerrojos a nivel del directorio, descrito en el capítulo 3.
- El código que dinámicamente traslada las variables cerrojo a bloques de memoria separados, con el fin de evitar la falsa compartición y medir de forma más precisa la mejora obtenida con el protocolo optimizado de acceso a cerrojos que hemos implementado.

4.2.1. Implementación del protocolo de encolado dinámico

Nuestra aproximación a la detección ideal de acceso a cerrojos asume que la instrucción de lectura-modificación-escritura presente en el ISA de SPARC, `ldstub`, se utiliza exclusivamente en el código de adquisición del cerrojo. Puesto que `ldstub` es una primitiva de sincronización, nunca forma parte del código de usuario, sólo del código de la librería de sincronización. La implementación de las llamadas `LOCK()` y `UNLOCK()`en la librería de RSIM se puede observar en la figuras 4.3 y 4.4, en el cual se utiliza el método test&test&set.[1]

[1]Las instrucciones `unimp` se utilizan como *ganchos* del simulador para obtener las estadísticas de tiempo empleado en estos fragmentos del código.

```
AcqTTS:
_AcqTTS:
     unimp 0x100c
     ldstub [%o0],%o1
AcqTTSretry:
     tst %o1
     be,a AcqTTSout
     membar #LoadLoad | #LoadStore
AcqTTSloop:
     ldub [%o0],%o1
     tst %o1
     bne AcqTTSloop
     nop
     ba AcqTTSretry
     ldstub [%o0],%o1
AcqTTSout:
     retl
     unimp 0x1000
```

Figura 4.3: Código de adquisición de cerrojo en la librería de RSIM.

```
RelTTS:
_RelTTS:
     unimp 0x100d
     membar #StoreStore | #LoadStore
     stb %g0,[%o0]
     retl
     unimp 0x1000
```

Figura 4.4: Código de liberación de cerrojo en la librería de RSIM.

A partir de aquí, es sencillo averiguar la dirección de memoria a la que accede la primitiva y marcar de algún modo las peticiones al directorio que acceden a dicha dirección. Nosotros hemos añadido código a nivel de L1 para interceptar las peticiones RMW y guardar la dirección del cerrojo al que se intenta acceder actualmente. Utilizando esta dirección es posible detectar posteriormente qué petición corresponde a la liberación del cerrojo, ya que en este sentido la escritura en el cerrojo no es diferente al resto. Así, tanto el mensaje de adquisición como el de liberación se marcan activando un determinado flag.

El controlador de directorio, observando dicho flag reconoce los accesos a un cerrojo y en función del tipo de petición que lanzó el procesador, es capaz de diferenciar qué mensajes corresponden a la adquisición del cerrojo (RMW) y cuáles a la liberación (*write*). En el directorio, la adición del código para el encolado dinámico de los peticionarios se ha situado en la función que determina las acciones de coherencia que se han de llevar a cabo para una cierta petición que llega al directorio. Ha sido necesario contemplar un nuevo escenario en el que el directorio recibe un mensaje de petición, lo procesa y como resultado no manda ninguna respuesta. Se ha implementado una estructura de datos para emular la caché del controlador de cerrojos, donde se guarda el vector de peticionarios. Junto a cada cerrojo se guardan las peticiones que llegan al directorio cuando el cerrojo está ocupado, y se recuperan una a una para

ser enviadas más tarde a su nodo origen, cuando dicho nodo es elegido como nuevo propietario del cerrojo.

Para simplificar la implementación del protocolo, los mensajes de adquisición y liberación del cerrojo enviados por los procesadores, así como las respuestas `LOCK_GRANTED` y `LOCK_RELEASED` por parte del directorio, no atraviesan las cachés de la misma forma que el resto: si bien pasan por el cauce para simular las latencias de acceso correctamente, ninguno de estos mensajes provoca cambios en el estado del bloque en caché, evitando reemplazos, writebacks o problemas con los registros MSHR. La comunicación entre procesadores y directorio se lleva a cabo por mensajes punto a punto.

También se ha librado al controlador de directorio de la responsabilidad de enviar dos mensajes (`LOCK_GRANTED` y `LOCK_RELEASED`) como respuesta a una sola petición (liberación sobre un candado solicitado). En su lugar, la L1 se encarga de duplicar el mensaje de liberación y enviar la petición original de vuelta al procesador para satisfacer el acceso de escritura inmediatamente. A continuación, la copia se envía hacia abajo por la jerarquía hasta el directorio, mediante comunicación punto a punto. Esta variación en la implementación del protocolo no tiene ningún efecto en los resultados, ya que la arquitectura simulada cuenta con *buffering* de almacenamientos: se permite que las instrucciones *store* se gradúen antes de su compleción, es decir, instrucción y dato se copian a un buffer de almacenamientos y se eliminan de la lista activa (ROB), de forma que su latencia no limita la llegada de nuevas instrucciones a la lista activa.

En este escenario de envío de mensajes punto a punto entre el procesador y el directorio, la única condición de carrera posible es el caso en que un nodo envía un mensaje de liberación y seguidamente otro de adquisición, y éste último llega antes al directorio que la liberación. En cualquier caso, no hay realmente condición de carrera: el directorio guarda la petición de adquisición puesto que el candado está ocupado, y cuando llega la liberación otorga permiso al siguiente procesador a la espera, en orden creciente de identificador.

4.2.2. Obtención de estadísticas sobre cada variable cerrojo

Una vez implementado en el simulador el mecanismo para identificar las direcciones de las variables cerrojo, es sencillo añadir soporte para recolectar datos acerca del uso de los cerrojos a lo largo de la ejecución de las aplicaciones. Así, hemos dotado al simulador de capacidad para extraer las siguientes estadísticas acerca de los cerrojos, para cada fase de la ejecución de la aplicación:

Por cerrojo:

- Número de procesadores que lo utilizaron.
- Tiempo medio de adquisición.

- Desviación típica del tiempo de adquisición.
- Número total de intentos de adquisición (RMWs).
- Número total de adquisiciones del cerrojo (ACQs).
- Número total de liberaciones del cerrojo (RELs).
- Promedio de RMWs por adquisición exitosa.
- Tasa de acierto de RMWs en la caché L2 (RMWs que se resolvieron en la L2).
- Tasa de acierto de ACQs en la caché L2 (RMWs que se resolvieron en la L2 y encontraron un cerrojo libre).
- Tasa de acierto de RELs en la caché L2.
- Número de RMWs recibidos en el directorio.
- Número de ACQs recibidos en el directorio (RMWs que llegaron hasta el directorio y encontraron un cerrojo libre).
- Número de RELs recibidos en el directorio.
- Número de SPINs recibidos en el directorio (peticiones de lectura, provocadas por la espera ocupada).

Por procesador:

- Número de cerrojos utilizados.
- Tiempo medio de adquisición.
- Desviación típica del tiempo de adquisición.
- Número de intentos de adquisición (RMWs) y media por cerrojo.
- Número de adquisiciones exitosas (ACQs) y media.
- Número de liberaciones (RELs) y media.
- Número de RMWs que acertaron en L2 y media.
- Número de ACQs que acertaron en L2 y media.
- Número de RELs que acertaron en L2 y media.
- Número de RMWs enviados al directorio y media.
- Número de ACQs enviados al directorio y media.
- Número de RELs enviados al directorio y media.
- Número de SPINs enviados al directorio y media.

4.2.3. Traslado de variables cerrojo a bloques con relleno

Para saber en qué medida el encolado dinámico de cerrojos beneficia el rendimiento de las aplicaciones paralelas creímos necesario eliminar cualquier factor que contribuyese a acrecentar la sobrecarga por sincronización de cerrojo. Así, surgió la necesidad de descartar los efectos de la falsa compartición entre cerrojos y otras variables del programa, mediante la utilización de relleno en los bloques cerrojo. Este relleno se puede conseguir muy fácilmente desde software: basta con modificar el código de la macro que contiene la declaración de un cerrojo, y añadir a cada lado de la variable cerrojo un array sin uso, de al menos el tamaño del bloque de memoria. La macro que contiene la declaración del cerrojo actualmente

```
define(LOCKDEC, ‘volatile int ($1);’)
```

pasaría a tener la siguiente forma

```
define(LOCKDEC, ‘int padd1($1) [LINESZ/sizeof(int)]; volatile int ($1);
int padd2($1) [LINESZ/sizeof(int)];’)
```

Sin embargo, como hemos comentado anteriormente, no nos ha sido posible recompilar las aplicaciones y obtener nuevamente binarios que funcionen sin problemas en el simulador. Por esta razón, hemos tenido que dedicar parte de nuestro trabajo a modificar el simulador para que dinámicamente coloque cada cerrojo en un bloque independiente. Esto lo hemos conseguido utilizando la función *our_shmalloc* con la que se simula la reserva de memoria compartida en RSIM. Ante el primer intento de adquisición de un cerrojo por parte de cualquier procesador, y sólo entonces, se reserva un bloque de memoria en el espacio compartido y se establece una correspondencia entre la dirección original del cerrojo y la que se obtiene con *our_shmalloc* en el momento del primer intento de adquisición. Antes de inyectar cualquier mensaje en el sistema de memoria, los procesadores chequean este *mapper* para ver si el acceso corresponde a un cerrojo, y si es así la petición que se introduce en el sistema de memoria lleva la nueva dirección, en lugar de la dirección original. Puesto que los mensajes del sistema de memoria se emparejan con sus respectivas instrucciones a través de etiquetas y no de la dirección de memoria accedida, el proceso es transparente al procesador.

4.3. Benchmarks

En esta sección se describen las características básicas de los cinco *benchmarks* científicos usados en nuestra evaluación. Este conjunto de aplicaciones es lo suficientemente variado para obtener unos resultados significativos. Las aplicaciones `BARNES`, `OCEAN`, `WATER-SP` y `WATER-NSQ` son del conjunto de *benchmarks* SPLASH-2 [WOT+95]. `UNSTRUCTURED` [MSH+95] es una aplicación de computación de fluidos dinámicos.

En la tabla 4.1 se muestran las aplicaciones utilizadas y el tamaño de problema elegido para cada una de ellas. El tamaño de problema empleado para cada aplicación es tal que la eficiencia paralela ($speedup(numprocs)/numprocs$) conseguida es

Aplicación	Tamaño
BARNES	8K partículas - 4 *timesteps*
OCEAN	258×258 celdas
UNSTRUCT	Mesh.2K
WATER-NSQ	512 moléculas
WATER-SP	512 moléculas

Tabla 4.1: Aplicaciones y tamaños utilizados en las simulaciones.

superior al 30 % para una configuración con 32 procesadores, según la evaluación llevada a cabo en [Ros04]. En todo caso, los resultados extraídos de las simulaciones corresponden a la fase paralela de las aplicaciones.

4.3.1. BARNES

La aplicación `BARNES` simula la interacción de un sistema de cuerpos (galaxias de partículas, por ejemplo) en tres dimensiones, sobre un cierto número de pasos de tiempo, utilizando el método jerárquico Barnes-Hut de N-cuerpos. Esta aplicación representa el dominio del problema como un árbol 8-ario con hojas que contienen información sobre cada cuerpo, y nodos internos que representan las celdas de espacio. La mayor parte del tiempo se ocupa en recorridos parciales del árbol (una por cuerpo) para calcular la fuerza ejercida sobre cuerpos individuales. Los patrones de comunicación son dependientes de la distribución de partículas y bastante desestructurados.

4.3.2. OCEAN

`OCEAN` estudia a gran escala los movimientos de un océano basándose en remolinos y sus propios límites. El algoritmo simula un recinto cúbico usando un modelo de corrientes discreto que tiene en cuenta la carga del viento debida a los efectos atmosféricos y la fricción con las paredes y el suelo del océano. El algoritmo realiza la simulación de muchos pasos hasta que se consigue un equilibrio de los remolinos y los flujos del océano. El trabajo que realiza en cada paso implica esencialmente resolver un conjunto de ecuaciones diferenciales parciales en el rango del espacio. Para ello, el algoritmo discretiza las funciones continuas mediante diferenciales finitas de segundo orden, estableciendo las ecuaciones diferenciales resultantes en cuadrículas de tamaño fijo que representan secciones horizontales del recinto del océano, y resuelve estas ecuaciones mediante un resolutor de ecuaciones que usa el algoritmo *multigrid red-back* de Gauss-Seidel. Cada tarea realiza los cálculos de la sección de la cuadrícula que le corresponde, normalmente comunicándose con otros procesos.

4.3.3. UNSTRUCTURED

`UNSTRUCTURED` es una aplicación de cálculo de fluidos dinámicos que utiliza una malla no estructurada para modelar una estructura física, como el ala o cuerpo de un avión. La malla es representada por nodos, aristas que conectan dos nodos y caras que conectan tres o cuatro nodos. La malla es estática y por tanto la conectividad no cambia. La malla es dividida espacialmente entre los diferentes procesadores usando un particionador recursivo que divide el cuerpo según sus coordenadas. Los cálculos se basan en una serie de bucles que iteran sobre los nodos aristas y caras. La mayoría de las comunicaciones ocurren entre las aristas y las caras de la malla.

4.3.4. WATER-NSQ

La aplicación `WATER-NSQ` realiza la simulación de un sistema molecular dinámico sobre un cuerpo N, es decir, con exceso de carga negativa, de las fuerzas y potenciales en un sistemas de moléculas de agua. Se usa para predecir algunas propiedades físicas del agua en estado líquido. Las moléculas están estáticamente distribuidas entre los procesadores, y la estructura de datos principal de esta aplicación es un gran *array* de registros que se usan para almacenar el estado de cada molécula. En cada paso del algoritmo los procesadores calculan la interacción de los átomos en cada molécula y las interacciones de unas moléculas con otras. Para cada molécula, el procesador que se encarga de su procesamiento calcula las interacciones con sólo la mitad de las moléculas que están delante de ella en el *array*, ya que las fuerzas entre las moléculas son simétricas, y por tanto cada par de interacciones entre dos moléculas es considerado sólo una. Entonces el estado asociado a las moléculas se actualiza. Aunque algunas partes del estado de la molécula son modificados en cada interacción, otras sólo cambian entre un paso y otro. La aplicación posee también una serie de variables que contienen la información de propiedades globales y son actualizadas continuamente por cualquier procesador.

4.3.5. WATER-SP

Esta aplicación resuelve el mismo problema que `WATER-NSQ`, pero utiliza un algoritmo más eficiente. Impone una cuadrícula de celdas uniforme tridimensional sobre el dominio del problema, y utiliza un algoritmo con complejidad temporal O(n), más eficiente que el de water-nsquared para un gran número de partículas. La ventaja de utilizar esta malla de celdas es que los procesadores que poseen una celda sólo necesitan mirar a las celdas del vecindario para encontrar moléculas que pueden estar dentro del radio de corte de moléculas en la caja que posee. El movimiento de moléculas dentro y fuera de las celdas causa que las listas sean actualizadas, resultando en comunicación entre procesadores.

Capítulo 5

Evaluación y resultados

En este capítulo realizamos un resumen de los resultados obtenidos experimentalmente a partir del simulador RSIM, modificado para soportar la política de encolado de peticionarios para líneas cerrojo. En primer lugar se muestran los parámetros utilizados a la hora de ejecutar las distintas aplicaciones en el simulador, a continuación los resultados más importantes obtenidos en dichas simulaciones junto con un análisis de los mismos, y finalmente una breve caracterización de la sincronización en cada una de las aplicaciones paralelas utilizadas.

5.1. Parámetros de simulación

La evaluación ha sido realizada simulando la ejecución de los *benchmarks* escogidos en un multiprocesador con las características que se muestran en la tabla 5.1. Cabe destacar algunos parámetros importantes, como por ejemplo el tiempo de acceso a memoria principal, fijado en 300 ciclos para analizar la repercusión del *memory wall* en la sobrecarga introducida por las actividades de sincronización. El número de ciclos del controlador de directorio también es digno de reseñar, ya que tiene un retardo de tan solo un ciclo, al estar situado en el chip del procesador.

5.2. Resultados

En esta sección se comentan los resultados obtenidos al introducir la nueva política de gestión aplicada sobre las líneas cerrojo, utilizando el mecanismo de encolado dinámico de peticionarios en el acceso a estas variables de sincronización. Con el fin de analizar la efectividad de la política de acceso a cerrojos por sí misma, hemos fijado el umbral de contención utilizado en todas las simulaciones en 0. Este parámetro establece el número de peticionarios simultáneos a partir del cual se activa la política

Sistema de 32 Nodos	
Parámetros del procesador ILP	
Tasa máxima *fetch/retire*	4
Ventana de instrucciones	64
Unidades funcionales:	
* Aritmética entera	2 unidades
* Punto flotante	2 unidades
* Generación de direcciones	2 unidades
Predictor de saltos:	
* Bits de historia	2 bits
* Contadores	512
* *Shadow mappers*	8
Parámetros de la caché	
Tamaño de la línea de caché	64 bytes
Cache L1 :	Escritura directa
* Tamaño	16KB
* Asociatividad	Mapeo directo
* Puertos de peticiones	2
* Tiempo de acierto	2 ciclos
Cache L2 :	Post-escritura
* Tamaño	64KB
* Asociatividad	4-vías
* Puertos de peticiones	1
* Tiempo de acierto	15 ciclos (6 + 9), segmentado
Número de MSHRs	8 por caché
Parámetros del directorio y de la memoria	
Ciclos del controlador de directorio (on-chip)	1 ciclo
Tamaño del buffer petic. pendientes	64 peticiones
Tamaño de la cabecera del paquete	16 bytes
Tiempos de creación de paquetes:	
* Primer mensaje de coherencia	4 ciclos
* Siguientes mensajes de coherencia	2 ciclos
Tiempo de acceso a memoria	300 ciclos
Factor de entrelazado de memoria	4
Parámetros del Bus	
Anchura	8 bytes
Frecuencia	1 ciclo
Latencia arbitraje	1 ciclo
Parámetros de Red	
Topología	Malla
Tamaño del *flit*	8 bytes
Latencia del *flit*	4 ciclos

Tabla 5.1: Parámetros básicos del sistema.

tipo cerrojo. Por tanto, al fijar su valor a 0 se consigue que todas las variables cerrojo se gestionen utilizando la política de encolado.

No obstante, la caracterización de la sincronización realizada para las cinco aplicaciones paralelas consideradas revela que es imprescindible utilizar un valor apropiado como umbral de contención, en lugar de tratar todos los bloques cerrojo por igual. Con

el umbral acertado es posible conseguir que el mecanismo de controlador de cerrojos propuesto se convierta en una solución general para lograr la sincronización eficiente, independientemente de las características de la sincronización en las aplicaciones.

El objetivo primordial de la utilización de una política alternativa para la gestión de las líneas de memoria, llevada a cabo por el controlador de directorio, es adaptarse a las particularidades de las variables de sincronización de tipo cerrojo con el fin de mejorar la escalabilidad y el rendimiento de las operaciones de `LOCK` y `UNLOCK` implementadas con un método tan sencillo como test&test&set. En esta sección utilizamos el *tiempo medio de adquisición de cerrojo* como métrica del rendimiento de este tipo de sincronización.

5.2.1. Tiempo de ejecución

El primer paso en nuestra evaluación es la obtención de datos acerca del rendimiento del método de sincronización original de RSIM y el peso de este componente en el tiempo de ejecución global de las aplicaciones utilizadas. El gráfico 5.1 resume de manera visual los resultados de los experimentos, mostrando claramente el porcentaje del tiempo total empleado en sincronización, distinguiendo la adquisición y liberación de cerrojos (*Acq* y *Rel*). Vemos como Barnes, Ocean y Unstruct dedican, respectivamente, el 20, 25 y 28 % de su tiempo de ejecución a la adquisición de cerrojos, mientras que en el caso de los dos Water la sobrecarga es algo menor, 9 % para Water-nsq y 16 % para Water-sp.

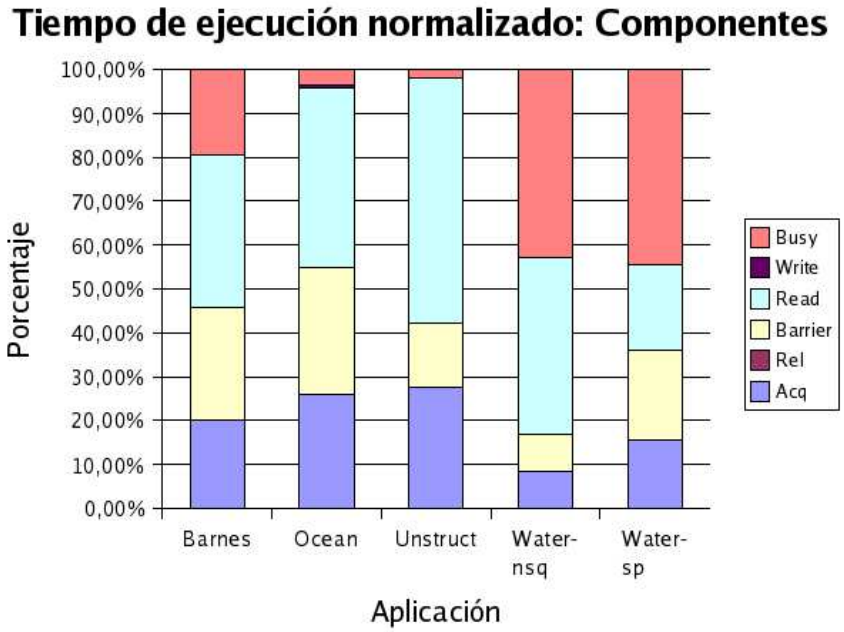

Figura 5.1: Tiempo de ejecución normalizado y componenentes.

Vemos como la liberación del cerrojo, que consiste en una simple instrucción de almacenamiento, no supone ninguna sobrecarga, pues la arquitectura simulada cuenta con *store buffering*: Las instrucciones de escritura se gradúan antes de su compleción, lo cual hace que la latencia del acceso no limite la llegada de nuevas instrucciones a la lista activa. Los *stores* se retiran del buffer de reordenación (ROB) y se colocan en el buffer junto con su dato hasta que el sistema de memoria completa el acceso, momento en el que se eliminan del todo. Por tanto estas instrucciones nunca son la causa de que el ROB no gradúe instrucciones a su tasa máxima por ciclo, y de ahí que su componente en el tiempo de ejecución global sea nulo.

También es notable el elevado tiempo de sincronización de barrera en Barnes, Ocean y Water-sp, en torno al 20-25 %, según cada caso. A primera vista, se puede pensar que no existe ninguna relación entre este tipo de sobrecarga de sincronización y la introducida por los cerrojos. Veremos más adelante con resultados experimentales cómo en realidad el tiempo de barrera depende en cierta medida del tiempo empleado en la adquisición de los cerrojos. Así, el tiempo de barrera empieza a contar desde el momento en que el primer procesador llega a la barrera, hasta que lo hace el último y todos son liberados de ella. Ahora bien, en fragmentos del código donde se produce la adquisición de un cerrojo inmediatamente antes de la llegada a la barrera, como sucede en Ocean, un método de adquisición ineficiente provoca, ante la llegada más o menos simultánea de muchos procesadores al cerrojo, que se alargue el tiempo que transcurre desde que el primero ejecuta la sección crítica hasta que lo hace el último. Esto se traduce directamente en un incremento del tiempo total que los procesadores emplean esperando en la barrera a que llegue el resto.

Con estos resultados iniciales, creemos prometedor el margen de mejora que se puede conseguir aplicando la política de manejo de bloques diseñada para hacer eficiente la transferencia del cerrojo en situaciones de elevada contención.

A efectos de la evaluación, hemos denominado *configuración base* a esta arquitectura de partida, que utiliza una sola política de gestión de líneas de memoria y se basa en el método test&test&set para la adquisición del cerrojo. Tomando estos resultados como referencia, llevamos a cabo una comparación entre esta arquitectura y aquella que emplea el controlador de cerrojos para optimizar la sincronización. A esta otra configuración que aplica una política especial a los bloques de memoria que contienen variables cerrojo la hemos denominado *configuración optimizada*.

El tiempo medio global de adquisición de cerrojo para una aplicación se calcula a partir de los tiempos medios de adquisición de cada cerrojo, sumando cada tiempo medio ponderado con su peso relativo en la aplicación. El peso de un cerrojo no es sino el número de adquisiciones sobre dicho cerrojo dividido entre el número total de adquisiciones de cerrojos en el programa.

Puesto que pretendemos eliminar cualquier factor ajeno al mecanismo de sincronización que contribuya a degradar el rendimiento de la configuración base, hemos repetido los experimentos con esta configuración utilizando el traslado dinámico de

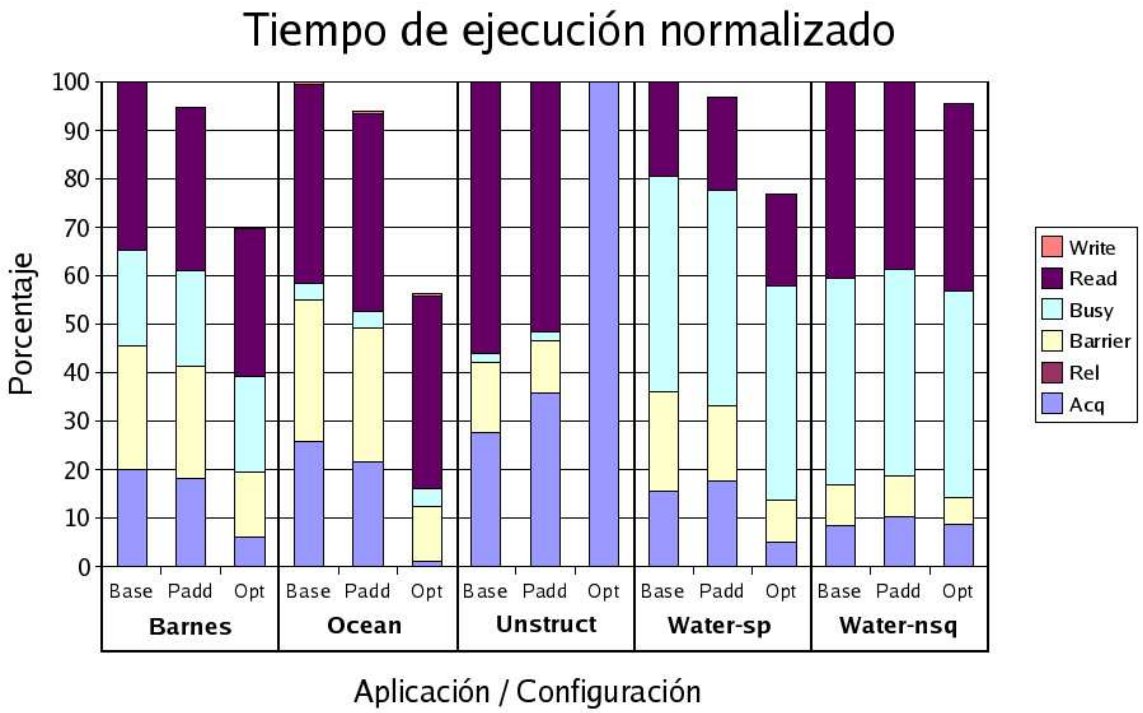

Figura 5.2: Componentes del tiempo de ejecución para cada configuración

cerrojos a bloques con relleno. Los resultados muestran que el pobre rendimiento de la sincronización por cerrojos en la arquitectura original no está motivado por problemas de falsa compartición entre bloques.

En la figura 5.2, que resume los principales resultados de nuestra evaluación, se observan los resultados de las tres configuraciones simuladas, correspondientes al caso base (*base*), al uso de bloques con relleno para evitar falsa compartición (*padd*) y al uso de la política de manejo de líneas optimizada para cerrojos (*opt*). En este gráfico se aprecia el tiempo de ejecución que obtenemos para cuatro de los cinco *benchmarks*, al utilizar una política de gestión de memoria especial para las líneas que contienen variables cerrojo. Vemos como en las aplicaciones con alta contención en el acceso a cerrojos la mejora obtenida es bastante significativa. El tiempo dedicado a la adquisición de cerrojos al aplicar la configuración optimizada pasa del 20 al 6 % en Barnes, del 25 al 2 % en Ocean, y del 16 al 6 % en Water-sp [1]. En Water-nsq apenas si hay diferencia entre ambas simulaciones. Por contra, en Unstruct, el componente debido a adquisición de cerrojo se multiplica: el tiempo total del caso *opt* no se muestra en el gráfico, es el 316 % del tiempo del caso base. Esto indica claramente que no todas las aplicaciones con uso intensivo de cerrojos se benefician de aplicar directamente la política de encolado a todos sus cerrojos, sino que es necesario establecer un umbral de contención por debajo del cual la gestión de las líneas cerrojo es convencional, dado que el rendimiento del método de test&test&set es prácticamente óptimo es escenarios de poca o ninguna contención.

[1] Porcentajes con respecto al tiempo del caso base de cada aplicación

Tiempo medio de adquisición			
Aplicación	Ciclos Base	Ciclos Opt.	Speedup
BARNES	14677	4414	3,32
OCEAN	255393	10446	24,44
UNSTRUCT	470	4344	0,11
WATER-NSQ	3294	3436	0,96
WATER-SP	192325	63406	3,03

Tabla 5.2: Tiempo medio de adquisición de cerrojo para cada configuración.

Reducción del tiempo de ejecución			
Aplicación	Ciclos Base	Ciclos Opt	Reducción
BARNES	3,96E+007	2,76E+007	30 %
OCEAN	1,34E+008	7,55E+007	44 %
UNSTRUCT	2,52E+008	7,96E+008	Aumento
WATER-NSQ	3,23E+007	3,09E+007	4 %
WATER-SP	1,35E+007	1,04E+007	23 %

Tabla 5.3: Reducción del tiempo de ejecución conseguida con la política de cerrojos.

En la tabla 5.2 mostramos el tiempo medio de adquisición para cada aplicación en ambas configuraciones simuladas, junto con el valor de aceleración obtenido. Para aplicaciones como Ocean, Barnes y Water-sp, la aceleración conseguida no se debe exclusivamente a la disminución del componente debido a cerrojos, sino que al reducir el tiempo de transferencia del cerrojo también se produce una reducción del tiempo de barrera. Nuestros experimentos muestran cómo en los *benchmarks* Ocean y Barnes se pasa de en torno al 25 % de tiempo de sincronización de barrera en la configuración base, a menos del 15 % al utilizar el controlador de cerrojos en las transferencias, y en Water-sp la reducción es similar, del 20 al 10 %. Esto contribuye enormemente a elevar el speedup obtenido con la utilización de la política de encolado dinámico de peticionarios del cerrojo.

5.3. Caracterización de la sincronización

En este apartado comentamos las características más significativas sobre la sincronización mediante cerrojos en cada una de las cinco aplicaciones bajo estudio. Este estudio es posible gracias a las estadísticas relativas a este tipo de variables que hemos conseguido extraer al modificar el simulador. La fase de ejecución considerada es siempre la paralela y el umbral de contención utilizado en el caso de la configuración optimizada es nulo en todo caso.

En esta caracterización de la sincronización utilizamos principalmente tres medidas cuantitativas sobre los cerrojos:

- *Tiempo medio de adquisición.* Es una medida del rendimiento de la sincronización. Se calcula como la media del tiempo transcurrido desde que el primer intento de adquisición se emite a la caché L1, hasta que se recibe la respuesta que concede la propiedad del candado.
- *Número medio de peticionarios.* Es una medida de la contención soportada por el cerrojo, utilizada en la configuración optimizada aprovechando la presencia de la cola de peticionarios. Se calcula como la media del número de bits activos en el vector de peticionarios en el momento de cada liberación.
- *Número medio de intentos por adquisición.* Es otra medida alternativa del grado de contención del cerrojo utilizada en la configuración base. Se calcula como el número de peticiones de adquisición (RMW) sobre dicho cerrojo, divido entre el número de adquisiciones exitosas conseguidas (ACQs).

Estas tres medidas particulares a un cerrojo se pueden convertir en globales ponderando el peso que cada cerrojo tiene en la ejecución de la aplicación. Así, el peso relativo de cada uno se establece en función de su número de adquisiciones en relación al número total de ACQs.

5.3.1. OCEAN

La fase de ejecución paralela de Ocean hace uso únicamente de dos cerrojos que son accedidos por los 32 procesadores del sistema. El 96 % del total de adquisiciones/liberaciones efectuadas se refieren al mismo cerrojo, en una proporción de 4128 frente a 192, con el problema considerado (258×258 celdas). El análisis de las estadísticas por procesador muestra que cada uno de los 32 procesos realiza el mismo número de adquisiciones del cerrojo en la fase paralela: 135.

Configuración base. Todos los procesadores utilizan los dos cerrojos y se observa que ambos sufren un elevado nivel de contención. Son necesarios, respectivamente, entre 11 y 15 intentos de media por cada adquisición exitosa. Esta es una de las razones por las que el tiempo medio de adquisición global en esta aplicación para la configuración base es tan alto, superior a 250.000 ciclos. Los datos también muestran que absolutamente todos los accesos RMW han de llegar hasta el directorio para ser satisfechos, esto es, ninguna adquisición del cerrojo se resuelve localmente en la caché del peticionario (no hay auto-transferencias). En cambio, un 18 % de las liberaciones acierta en la caché L2, gracias a que la sección crítica se completa antes de que la copia en propiedad del bloque sea invalidada por un intento de adquisición

fallido procedente de otro procesador. En cuanto al tráfico causado por la espera ocupada, el directorio ha recibido un total de 47956 peticiones de lectura del cerrojo, a una media de 10,88 y 15,81 por adquisición, según el cerrojo.
Los datos son similares en prácticamente todos los procesadores, si bien puntualmente vemos que alguno consigue un tiempo de adquisición mucho menor (15.000 ciclos) dado que por las circunstancias de la ejecución necesita sólo 1,55 RMWs de media por adquisición, y un 93 % de las liberaciones aciertan en caché. El resto de procesadores tiene estadísticas muy parecidas, con tiempos de adquisición que oscilan entre 200.000 y 300.000 ciclos, y una media de 11,27 RMWs/ACQ.

Configuración optimizada. Al poner en funcionamiento el encolado dinámico de los peticionarios del cerrojo, todas los intentos de adquisición se convierten en exitosos, así que la media es de un RMW por ACQ. Por esta razón, no se recibe ninguna petición de lectura en el directorio, al no tener lugar la espera ocupada. Obviamente, el controlador de cerrojos requiere que todos los mensajes (RMW, ACQ y REL) pasen por el directorio. El tiempo medio de adquisición global conseguido por esta configuración es de poco más de 10.000 ciclos, frente a los más de 250.000 de la configuración base. En el caso del cerrojo más utilizado, el número medio de peticionarios es de 8,63, mientras que en el segundo cerrojo aumenta hasta los 14,53. Un valor tan cercano a 16 revela que prácticamente los 32 procesadores del sistema llegan al mismo tiempo a la sección crítica y compiten simultáneamente por la adquisición de este cerrojo.

5.3.2. BARNES

Con un tamaño del problema de 8K partículas, Barnes utiliza 77 cerrojos en su fase paralela, de los cuales sólo 3 son utilizados por los 32 procesadores, mientras que otros 9 o 10 cerrojos son accedidos por entre 15 y 26 procesadores. Esta docena de cerrojos aglutina el 40 % de todas las adquisiciones de la aplicación (casi 7.000 de 17408). Observando los datos por procesador vemos que cada procesador utiliza en promedio 21 cerrojos, siendo el mínimo 16 y el máximo 32 cerrojos. El número de adquisiciones varía entre 400 y 800 según el procesador, y la media está en 544.

Configuración base. Para los 12 cerrojos más utilizados de la aplicación, el número medio de intentos por adquisición es de 1,64 (11.487 RMWs para 6.980 ACQs), aunque la media global es de 1,28 RMWs/ACQ. El tiempo medio de adquisición global es de 14.677 ciclos, si bien en los cerrojos más usados sube hasta casi 65.000 ciclos. Vemos que un 45 % los RMW aciertan en L2, al contrario que en Ocean, donde este porcentaje es del 0 %. De los intentos de adquisición exitosos, el 58 % encuentran el cerrojo libre en su caché, mientras que las liberaciones llegan al 66 % de acierto en la L2. Esto revela que en Barnes hay un buen número de autotransferencias del cerrojo. La espera ocupada causa el envío de 10.931 mensajes al directorio, a una media de 0,63 mensajes por cerrojo. Por procesadores, no hay gran

diferencia entre unos y otros, estando el tiempo medio de adquisición entre 6.000 y 25.000 ciclos.

Configuración optimizada. En este caso, el tiempo medio de adquisición global se reduce a una tercera parte del valor base: 4414 ciclos. El número medio de peticionarios global es de 1,11, aunque para los diez cerrojos utilizados por más procesadores la media se sitúa en 2,36.

5.3.3. WATER-NSQ

Para esta aplicación y el tamaño de problema considerado (512 partículas), se utiliza un total de 516 cerrojos en la fase paralela. Salvo 20 de ellos, los otros 496 cerrojos son utilizados por 17 procesadores y adquiridos exactamente 51 veces cada uno. En total, durante la ejecución se llevan a cabo 26576 adquisiciones del cerrojo.

Configuración base. Los 496 cerrojos mencionados suponen el 95 % del total de adquisiciones en la aplicación. El tiempo medio de adquisición de este grupo de cerrojos es de tan solo 77 ciclos, dado que una buena parte de ellos tiene una media de 17 ciclos. En estos casos, los 51 intentos de adquisición de cada cerrojo aciertan en caché L2 y encuentran un valor del candado libre (1 RMW/ACQ). Las liberaciones también se resuelven localmente salvo excepciones. Sin embargo, el tiempo medio de adquisición global para esta configuración es de 3294 ciclos, provocado por la influencia de dos cerrojos que no pertenecen a este grupo y que son utilizados por los 32 procesadores. Su tiempo de adquisición es mucho más elevado por el efecto de la contención (400.000 y 163.000 ciclos, respectivamente) aunque su peso en el tiempo global es pequeño ya que sobre ellos sólo se realizan 96 y 288 adquisiciones (15,5 y 5,9 RMWs/ACQ). A pesar esto, sus elevados valores del tiempo de adquisición contribuyen a aumentar el tiempo medio global, desde 80 a casi 3300 ciclos. Al observar las estadísticas por procesador vemos que el uso de los cerrojos es uniforme: 275 cerrojos usados por cada uno, con 829-832 adquisiciones y promedios de acierto en L2 del 82 %, 92 % y 67 % para RMWs, ACQs y RELs, respectivamente.

Configuración optimizada. El efecto de la aplicación de la política de encolado sobre todos los cerrojos tiene un efecto curioso en Water-nsq. Como se puede intuir, los 496 cerrojos que por lo general resolvían sus accesos de forma local, ahora ven penalizado su rendimiento pues cada adquisición ha de llegar hasta el directorio del nodo *home* y regresar al nodo origen para completar la autotransferencia. Esto hace que ahora todos estos cerrojos tengan tiempos de adquisición que oscilan entre los 2000 y los 4000 ciclos, frente a los 17 de la configuración base. Ahora bien, la política de encolado dinámico de peticionarios elimina por completo la contención soportada por los dos cerrojos mencionados anteriormente, y gracias a ello reducen su tiempo

medio de adquisición de 400.000 y 163.000, a 36.000 y 49.000 ciclos, respectivamente. La eliminación del cuello de botella que constituyen estos dos cerrojos compensa la degradación sufrida por el resto debido a la ausencia de un umbral de contención adecuado. Así, finalmente el tiempo medio de adquisición global (3436) es apenas 140 ciclos mayor en esta configuración que en el caso base. Por tanto, nuestra caracterización revela que en aplicaciones como Water-nsq, donde la auto-transferencia del cerrojo es habitual, el controlador de cerrojos debe limitar su actuación a los cerrojos verdaderamente contendidos. Con el umbral de contención adecuado, el controlador de cerrojos es capaz de suavizar el efecto de los dos cerrojos altamente contendidos al tiempo que deja intacto el funcionamiento de los otros 514 cerrojos, reduciendo el tiempo dedicado a este tipo de sincronización a niveles mínimos.

5.3.4. WATER-SP

A diferencia de Water-nsq, el algoritmo utilizado en Water-sp utiliza únicamente cuatro cerrojos para resolver el mismo problema. Con un tamaño de problema de 512 partículas, Water-sp realiza 352 adquisiciones de cerrojos, distribuidas entre los cuatro cerrojos en proporciones 64/64/192/32. Los 32 procesadores del sistema utilizan los cuatro cerrojos. Cada proceso realiza en total 11 adquisiciones en la fase paralela de su ejecución.

Configuración base. Puesto que todos los procesos compiten por los cuatro cerrojos, todos ellos presentan un nivel de contención bastente significativo, de ahí que sus tiempos medios de adquisición estén por encima de los 11.000 ciclos. Un aspecto un tanto peculiar que observamos es que los dos cerrojos con mayor tiempo medio de acceso (276.000 y 387.000 ciclos) presentan un número de intentos medio por adquisición inferior a 10 RMWs/ACQ, y sin embargo sus tiempos de adquisición son desorbitados. Probablemente esto se deba a la longitud y duración de las secciones críticas protegidas por dichos cerrojos, pues cuanto más tiempo se mantiene un cerrojo en propiedad, mayor es el tiempo de adquisición de los procesos a la espera, y por tanto, mayor es la media. En cualquier caso, estas considerables magnitudes hacen que el tiempo medio global llegue a los 192.325 ciclos.

Configuración optimizada. Con esta configuración conseguimos reducir el tiempo medio de adquisición a una tercera parte, pero todavía son unos nada despreciables 63585 ciclos. Esto da una idea de que efectivamente la duración de las secciones críticas en esta aplicación es relativamente larga, pues el tiempo de adquisición del cerrojo depende no sólo del tiempo de transferencia del peticionario al propietario, sino también del tiempo que el cerrojo está en propiedad de cada proceso.

5.3.5. UNSTRUCT

Hemos visto en la sección de resultados que esta aplicación, pese a realizar un uso intensivo de cerrojo, no se beneficia de la aplicación de una política especial para la transferencia de los bloques cerrojo entre procesadores. A continuacion caraterizamos la sincronización en Unstruct con el fin de averiguar el porqué de la degradación de su rendimiento cuando utilizamos el controlador de cerrojos.
A primer vista obsevamos que el número de cerrojos utilizados en este *benchmark* es muy elevado, con un total de 2800 variables de este tipo. Esto sucedía en menor medida en Water-nsq (516 cerrojos), pero por el contrario en dicha aplicación no observamos semejante empeoramiento del rendimiento de la sincronización. La explicación a este hecho radica en una diferencia fundamental entre Unstruct y Water-nsq: el número de adquisiciones total es muchísimo mayor en el primero de ellos, de casi 4.700.000, a una media de unas 150.000 por procesador, frente a las 51 ACQ/procesador de Water-nsq. Estos datos preliminares nos llevan a pensar que el peso de la adquisición de cerrojos en el tiempo total de ejecución (28 %) no se debe a la ineficiencia de las operaciones de sincronización, sino a la propia abundancia de éstas en la ejecución del programa. Los cerrojos son utilizados por entre 4 y 20 procesos, estando el promedio en 13,22. Cada proceso utiliza una media de 1157 cerrojos en la resolución del problema, en este caso una malla de 2K nodos.

Configuración base. El tiempo medio de adquisición cuando no se utiliza el controlador de cerrojos es tan solo de 470 ciclos, con una pequeña desviación estándar de 200 ciclos de media. El 99 % de los intentos de adquisición encuentran un cerrojo libre, lo cual da revela que la contención es prácticamente nula en todos los cerrojos. Otro indicador de este hecho es que el promedio de lecturas del cerrojo recibidas en el directorio es de 11 por cerrojo, lo cual es despreciable teniendo en cuenta que cada cerrojo se adquiere de media unas 1675 veces. Pese a que sólo el 31 % de los ACQs y RELs aciertan en L2, el tiempo de transferencia se mantiene reducido al no existir competencia por los bloques.

Configuración optimizada. Los datos de esta configuración confirman los niveles de contención anteriores: el número medio de peticionarios del cerrojo es de 0,01, en plena concordancia con la medida del caso base (1,01 RMWs/ACQ). Al activar la política de cerrojo incondicionalmente, ese 31 % de adquisiciones que se resolvía en 17 ciclos (2 ciclos para acceder a L1 y 15 para L2) ahora se ve obligado a pasar por el directorio una y otra vez, haciendo que este millón y medio de ACQs multipliquen su tiempo de adquisición por 30. Puesto que esta degradación afecta a una tercera parte del total de ACQs, el factor de aumento del tiempo medio de adquisición global es de 10, pasando de 470 a 4344 ciclos. El efecto en el tiempo de ejecución total es proporcional al peso del componente de cerrojos (28 %), lo cual corresponde aproximadamente con los resultados obtenidos en la sección anterior: el tiempo de la configuración óptima es el 300 % del tiempo en la configuración base.

Capítulo 6

Conclusión y trabajo futuro

En este capítulo aglutinamos las principales ideas que se derivan de la realización de este proyecto y proponemos nuevas vías de investigación que se pueden emprender partiendo de un mecanismo de sincronización como es el controlador de cerrojos.

6.1. Conclusión

En este proyecto hemos realizado un estudio pormenorizado de la sincronización mediante cerrojos en arquitecturas de memoria compartida escalables, como son los multiprocesadores cc-NUMA. Hemos mostrado el problema de escalabilidad que tienen los métodos de adquisición sencillos cuando el número de procesadores que utilizan el cerrojo es elevado, y establecido una relación directa entre el peso de la sobrecarga introducida por las actividades de sincronización y la latencia de acceso a memoria.

Como una solución posible al problema de la sincronización para los multiprocesadores con protocolo de coherencia de caché basados en directorio, hemos desarrollado un mecanismo denominado *controlador de cerrojos*, mediante el cual es posible mantener el método de adquisición del cerrojo sencillo y al mismo tiempo hacer eficiente la adquisición en situaciones de elevada contención, donde normalmente el rendimiento de las soluciones software más simples sufre una fuerte degradación.

Gracias a este hardware, es posible beneficiarse de las ventajas de los métodos sencillos para los cerrojos con poca contención, como es su reducida latencia, al tiempo que se hace eficiente la transferencia de otros cerrojos en los que un elevado número de procesadores compite por su acceso. Esto se traduce en una reducción del impacto de la sincronización en el tiempo de ejecución de aquellas aplicaciones que hacen un uso frecuente de este tipo de variables para proteger las secciones críticas de su código.

En este trabajo se ha descrito en detalle tanto el funcionamiento como la imple-

mentación del controlador de cerrojos, y se ha evaluado el rendimiento de la política de gestión de líneas de memoria aplicada por dicho mecanismo sobre los bloques que contienen cerrojos. Para llevar a cabo la evaluación de nuestra propuesta se ha utilizando la herramienta de simulación RSIM y cinco *benchmarks* científicos que representan diferentes tipos de aplicaciones paralelas con características de sincronización variadas. Como parte de nuestro estudio, hemos llevado a cabo una caracterización de la sincronización mediante cerrojo en cada una de las cinco aplicaciones, como complemento a la evaluación del controlador de cerrojos.

Los resultados obtenidos indican una reducción del tiempo de ejecución muy significativa en tres de los cinco *benchmarks* utilizados. Esta reducción es del 44 % en el caso de Ocean, el 30 % en Barnes y el 23 % en Water-sp, mientras que para Water-nsq es más suave, de tan sólo el 4 %. Estos datos revelan que el controlador de cerrojos, desde su posición junto al controlador de directorio, tiene el potencial de mejorar en gran medida el tiempo de ejecución de aquellas aplicaciones cuyos cerrojos se encuentran en un estado de media o elevada contención.

Por otro lado, merced a la existencia de un umbral de contención, este mecanismo tiene la capacidad de evitar la puesta en marcha de la política de transferencia del cerrojo desde el directorio, cuando se trata de cerrojos auto-transferidos o muy poco contendidos, ya que en tal caso dicha política afecta negativamente al rendimiento. Nuestra evaluación muestra que la aplicación Unstructured sufre un aumento del tiempo de ejecución superior al 300 % al utilizar un umbral de contención nulo. Esto pone de manifiesto la necesidad de utilizar un valor apropiado para este parámetro con el fin de conseguir hacer de este mecanismo una solución general para lograr la sincronización eficiente.

En resumen, en este trabajo hemos presentado y evaluado el controlador de cerrojos, un mecanismo hardware acoplado al controlador de directorio que mejora la eficiencia de las operaciones de sincronización en los multiprocesadores cc-NUMA mediante la detección en hardware del nivel de contención de cada cerrojo y la aplicación selectiva de una política de encolado dinámico de los peticionarios del cerrojo.

6.2. Trabajo futuro

En esta sección proponemos algunas líneas de investigación que podrían iniciarse en el futuro a raíz del estudio llevado a cabo en este proyecto:

En primer lugar, constituiría el siguiente paso en el desarrollo de esta propuesta efectuar un estudio experimental acerca del valor óptimo del umbral de contención que debe utilizar el controlador de cerrojos para evitar aplicar la política de encolado dinámico de cerrojos en casos en los que la política de coherencia convencional da un resultado óptimo en la transferencia del cerrojo. O incluso podría estudiarse la posibilidad de extender el controlador para convertir el umbral en un valor adaptativo

propio de cada cerrojo, en lugar de ser un parámetro del sistema.

Otro posible trabajo que tendría como meta contrastar la validez de este estudio sería llevar a cabo una comparación del rendimiento del mecanismo que hemos propuesto con otras implementaciones del método de adquisición. Así, podría evaluarse la sobrecarga introducida con algoritmos software más avanzados, como es el caso de los cerrojos basados en array o las listas de peticionarios del cerrojos, y comparar estos resultados con los que se obtienen con este esquema de algoritmo de adquisición sencillo apoyado por un hardware de controlador de cerrojos.

Una línea de desarrollo paralela que surge también a partir de este proyecto es el diseño y evaluación de un mecanismo de predicción de acceso a cerrojos, mediante el cual el hardware pueda inferir qué líneas de memoria contienen variables cerrojo y marcarlas, con el fin de que mecanismos como el controlador de cerrojos utilicen esta información para extraer un mayor rendimiento.

Por último, pero no por ello menos importante, sería muy interesante intentar aplicar y evaluar esta propuesta en una arquitectura CMP (*Chip Multiprocessor*), ya que a partir de un cierto número de núcleos integrados en el chip, se requiere una red punto a punto para la comunicación y por tanto estos sistemas necesitan una estructura de directorio para implementar el protocolo de coherencia.

Bibliografía

[AG89] George S. Almasi and Alan Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings, Redwood CA, 1989.

[AG03] Manuel E. Acacio and José M. García. Techniques for improving the performance and scalability of directory-based shared-memory multiprocessors: A survey. *Journal of Computer Science and Technology, Vol. 3, No. 2*, pages 1–8, October 2003.

[AGGD01] Manuel E. Acacio, José González, José M. García, and José Duato. A new scalable directory architecture for large-scale multiprocessors. In *Proc. of the 7th International Symposium on High Performance Computer Architecture (HPCA-7)*, pages 97–106, January 2001.

[AGGD02] Manuel E. Acacio, José González, José M. García, and José Duato. A novel approach to reduce l2 miss latency in shared-memory multiprocessors. In *Proc. 16th Int'l Parallel and Distributed Processing Symp.*, April 2002.

[AGGD04] Manuel E. Acacio, José González, José M. García, and José Duato. An architecture for high-performance scalable shared-memory multiprocessors exploiting on-chip integration. *IEEE Transactions on Parallel and Distributed Systems, Vol. 15, No. 8*, pages 755–768, August 2004.

[AGGD05] Manuel E. Acacio, José González, José M. García, and José Duato. A two-level directory architecture for highly scalable cc-numa multiprocessors. *IEEE Transactions on Parallel and Distributed Systems, Vol. 16, No. 1*, pages 67–79, January 2005.

[ALM01] Bilge S. Akgul, Jaehwan Lee, and Vincent J. Mooney. A system-on-a-chip lock cache with task preemption support. In *CASES*, pages 149–157, 2001.

[And90] Thomas E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems 1, Vol. 1*, pages 6–16, January 1990.

[ANMB97] Ernest Artiaga, Nacho Navarro, Xavier Martorell, and Yolanda Becerra. Implementing parmacs macros for shared memory multiprocessor environments. Technical report 07, Polytechnic University of Catalunya, Department of Computer Architecture, January 1997.

[BW04] Bradford M. Beckmann and David A. Wood. Managing wire delay in large chip-multiprocessor caches. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 319–330, May 2004.

[CS99] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[DEC98] DEC. *Alpha Architecture Handbook Version 4*, October 1998.

[FG05] Ricardo Fernández and José M. García. RSIM x86: A cost-effective performance simulator. In *19th European Conference on Modelling and Simulation*, pages 774–779, Riga, Latvia, June 2005. European Council for Modelling and Simulation.

[GT90] Gary Graunke and Shreekant Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer, Vol. 23*, pages 60–69, June 1990.

[GVW89] James Goodman, Mary K. Vernon, and Philip J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 64–75, April 1989.

[HM93] Maurice Herlihy and Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–301, May 1993.

[HPRA02] Christopher J. Hughes, Vijay S. Pai, Parthasarathy Ranganathan, and Sarita Adve. Rsim: Simulating shared-memory multiprocessors with ilp processors. *IEEE Computer*, 35(2), February 2002.

[IBM98] IBM. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, 1998.

[KBG97] Alain Kägi, Douglas C. Burger, and James R. Goodman. Efficient synchronization: let them eat qolb. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 170–180, June 1997.

[KJCS99] Sanjeev Kumar, Dongming Jiang, Rohit Chandra, and Jaswinder Pal Singh. Evaluating synchronization on shared address space multiprocessors: Metodology and performance. In *Proc. of the 1999 ACM SIGMETRICS Conf., Atlanta (USA)*, pages 23–34, May 1999.

[KSS96] Tareef S. Kawaf, D. John Shakshober, and David C. Stanley. Performance analysis using very large memory on the 64-bit alphaserver system. 1996.

[Kä99] Alain Kägi. *Mechanisms for efficient shared-memory lock-based synchronization*. PhD thesis, CS Dept., Univ. of Wisconsin-Madison, 1999.

[LA94] Beng-Hong Lim and Anant Agarwal. Reactive synchronization algorithms for multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 25–35, October 1994.

[LLJ+92] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The stanford dash multiprocessor. *IEEE Computer, 25(3)*, pages 63–79, 1992.

[LLS06] Tong Li, Alvin R. Lebeck, and Daniel J. Sorin. Spin detection hardware for improved management of multithreaded systems. *IEEE Transactions on Parallel and Distributed Systems, vol.17, no. 6*, pages 508–521, 2006.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared memory multiprocessors. *ACM Transactions on Computer Systems, Vol. 9*, pages 21–65, February 1991.

[MLH94] Peter Magnusson, Anders Landin, and Erik Hagersten. Efficient software synchronization on large cache coherent multiprocessors. SICS Research Report T94:07, Swedish Institute of Computer Science, Kista, Sweden, February 1994.

[MSH+95] Shubhendu S. Mukherjee, Shamik D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers, and Joel Saltz. Efficient support for irregular applications on distributed-memory machines. *Proc. of the 5th Int'l Symposium on Principles & Practice of Parallel Programming (PPOPP'95)*, pages 68–79, July 1995.

[MT01] José F. Martínez and Josep Torrellas. Speculative locks for concurrent execution of critical sections in shared-memory multiprocessors. In *Workshop on Memory Performance Issues*, June 2001.

[MT02] José F. Martínez and Josep Torrellas. Speculative synchronization: Applying thread-level speculation to parallel applications. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.

[PRA97] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. *RSIM Reference Manual, version 1.0*. Rice University, Electrical and Computer Engineering Department, August 1997.

[RAG05] Alberto Ros, Manuel E. Acacio, and José M. García. A novel lightweight directory architecture for scalable shared-memory multiprocessors. In José C. Cunha and Pedro D. Medeiros, editors, *11th International Euro-Par Conference*, volume 3648, pages 582–591, Lisbon (Portugal), August 2005. Springer-Verlag.

[Raj02] Ravi Rajwar. *Speculation-Based Techniques for Transactional Lock-Free Execution of Lock-Based Programs*. PhD thesis, CS Dept., Univ. of Wisconsin-Madison, 2002.

[RG01] Ravi Rajwar and James R. Goodman. Speculative lock elison: Enabling highly concurrent multithreaded execution. In *34th International Symposium on Microarchitecture*, pages 294–305, December 2001.

[RKG04] Ravi Rajwar, Alain Kägi, and James R. Goodman. Inferential queueing and speculative push. *Int. J. Parallel Program.*, 32(3):225–258, 2004.

[RL96] Umakishore Ramachandran and Joonwon Lee. Cache-based synchronization in shared memory multiprocessors. *Journal of Parallel and Distributed Computing, Vol. 32*, pages 11–27, May 1996.

[Ros04] Alberto Ros. Diseño y evaluación de una arquitectura basada en directorio ligero. Master's thesis, Universidad de Murcia, Departamento de Ingeniería y Tecnología de Computadores, 2004.

[RS84] Larry Rudolph and Zary Segall. Dynamic decentralized cache schemes for mimd parallel processors. In *11th Annual International Symposium on Computer Architecture*, pages 340–347, June 1984.

[RS02] Peter Rundberg and Per Stenstrom. Reordered speculative execution of critical sections. Technical report 02-07, Chalmers University of Technology, Department of Computer Engineering, Goteborg, Sweden, February 2002.

[SGI98] SGI. *MIPS R10000 Microprocessor User's Manual Version 2.0*. Mountain View, CA., 1998.

[Wal91] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 26, pages 176–189, New York, NY, 1991. ACM Press.

[WOT+95] Steven C. Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. pages 24–36, June 1995.