UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Planificación de tareas considerando la contención en la jerarquía de memoria de procesadores multinúcleo

Tesina de Máster en Ingeniería de Computadores
Curso 2011/2012

Departamento de Informática de Sistemas y Computadores

Josué Feliu Pérez

Directores:
Julio Sahuquillo Borrás
Salvador V. Petit Martí

Julio de 2012

Índice general

1. Introducción **8**

2. Estado del arte **11**

2.1. Estado del arte relacionado con la contención en el acceso a memoria principal 11

2.2. Estado del arte relacionado con la contención en el acceso a la cache LLC 13

3. Plataforma experimental **15**

3.1. Descripción del sistema 15

3.2. Características relevantes de la arquitectura 16

3.2.1. Contadores de prestaciones 16

3.2.2. *Huge pages* 18

4. Características de la LLC: geometría y arquitectura **22**

4.1. Motivación 23

4.2. Trabajo previo 25

4.3. Diseño de los tests experimentales 26

4.3.1. Obtención del tamaño de la línea de cache 26

4.3.2. Obtención del número de conjuntos y vías de la cache 28

4.3.3. Obtención del número de caches y su grado de compartición . . 30

5. Análisis de la degradación de prestaciones **32**

5.1. Caracterización de los benchmarks 32

5.2. Diseño del microbenchmark básico 34

5.3. Degradación debida a contención en el acceso a memoria principal . . . 35

5.3.1. Degradación variando el ancho de banda consumido por los co-runners 36

5.3.2. Degradación variando el número y ubicación de los co-runners . 37

5.3.3. Degradación del IPC para un ancho de banda total consumido . 39

5.4. Degradación debida a contención en el acceso a la cache L2 40

6. Algoritmos de planificación propuestos **42**
6.1. Algoritmo base: planificación considerando el ancho de banda de acceso a memoria principal 42
6.2. Planificación considerando el ancho de banda de la jerarquía de memoria 43
6.3. Planificación considerando la degradación del IPC 46

7. Evaluación del planificador propuesto **50**
7.1. Diseño de las cargas y metodología de la evaluación 50
7.2. Prestaciones de las propuestas 52

8. Conclusiones, Publicaciones y Trabajo Futuro **58**
8.1. Conclusiones 58
8.2. Publicaciones 59
8.3. Trabajo Futuro 60

Bibliografía **61**

Índice de figuras

3.1. Jerarquía de memoria del Intel Xeon X3320 y del IBM Power5 16
3.2. Traducción de direcciones para una LLC de 4 MB utilizando páginas de 4 KB 19
3.3. Traducción de direcciones para una LLC de 4 MB utilizando *huge pages* de 2 MB 20

4.1. Degradación del IPC de *bzip2* 24
4.2. Degradación del IPC de *bzip2* variando la ubicación del microbenchmark y su BTR_{MM} 24
4.3. Pseudocódigo para determinar la longitud de la línea de cache 26
4.4. Tasa de acierto global en la cache variando la longitud de la línea de la matriz 27
4.5. Pseudocódigo para determinar el número de conjuntos y vías de la cache 28
4.6. Tasa de acierto global en la cache variando el número de accesos diferentes y el *stride* 29
4.7. Tasa de acierto media en la cache ejecutando dos instancias del algoritmo de llenado de conjuntos en diferentes pares de núcleos 30

5.1. IPC para cada benchmark 33
5.2. BTR_{L2} para cada benchmark 33
5.3. BTR_{MM} para cada benchmark 34
5.4. Pseudocódigo del microbenchmark 35
5.5. Degradación del IPC variando el BTR_{MM} de los co-runners 36
5.6. Ubicación de los procesos. Leyenda: B (Benchmark) M (Microbenchmark) 38
5.7. Degradación del IPC variando el número y ubicación de los co-runners 38
5.8. Degradación del IPC sufrida por los benchmarks con un BTR_{MM} total de 30 trans/usec 39
5.9. Degradación del IPC variando el BTR_{L2} de los co-runners 41

7.1. Aceleración respecto a la planificación de Linux 53
7.2. Diferencia del BTR_{L2} entre las dos caches 53

7.3. Media y varianza de la diferencia del BTR_{L2} entre las dos caches 54
7.4. Evolución de la diferencia del BTR_{L2} los primeros 160 *quanta* de ejecución 55
7.5. Comparación de la diferencia del BTR_{L2} con los diferentes algoritmos de planificación 56
7.6. Porcentaje de las aplicaciones que utilizando el planificador PcD presentan una aceleración/deceleración del IPC en cada intervalo respecto al planificador PcJ 57

Índice de tablas

3.1. Especificaciones del sistema 15
3.2. Eventos hardware monitorizados con los contadores de prestaciones . . 17

7.1. Cargas 51

Capítulo 1

Introducción

Los sistemas multinúcleo se han convertido en la opción común utilizada para el diseño de microprocesadores de altas prestaciones. Estos multiprocesadores en chip (CMP) incorporan varios núcleos en un mismo chip y tienen potencial para ofrecer unas prestaciones superiores a las de los procesadores mononúcleo reduciendo, además, los problemas de consumo, encapsulado y refrigeración que éstos presentaban.

La mayoría de estos CMPs son multiprocesadores simétricos (SMP), lo que significa que los diversos núcleos que forman el chip acceden a una memoria principal compartida. Como resultado, la red de interconexión entre los núcleos y la memoria principal es el principal cuello de botella que presentan estos procesadores. En los primeros CMPs, la componente más importante del cuello de botella era la latencia de acceso a memoria principal. Sin embargo, a medida que las capacidades multinúcleo y multihilo de los procesadores han ido creciendo, la contención en el ancho de banda disponible para acceder a memoria se ha convertido en la principal preocupación, pues además de reducir las prestaciones de los CMPs complica su escalabilidad con el número de núcleos e hilos de ejecución.

Cuando el número de trabajos a ejecutar supera el número de núcleos, las estrategias de planificación que consideran el ancho de banda disponible en el acceso a memoria pueden ayudar al planificador a reducir la contención en el sistema, evitando ejecutar de manera concurrente aplicaciones con unos altos requisitos de ancho de banda [XWY10] [ANP03] [ANP04]. Dado que el ancho de banda requerido por las aplicaciones puede variar drásticamente durante su ejecución, estas estrategias de planificación monitorizan en tiempo de ejecución los requisitos de los procesos para predecir la demanda de ancho de banda esperada en próximos *quantums*. A partir de esta información, se planifican los procesos a ejecutar para alcanzar una utilización alta del bus, evitando infrautilizar un recurso importante, pero tratando que esta alta utilización no exceda la capacidad del bus, lo que provocaría contención en el acceso a memoria y reduciría las prestaciones de los procesos en ejecución. Como se describirá en el Capítulo 2, muchos

trabajos de investigación se han centrado en encontrar el mejor compromiso entre alta utilización y baja contención en el bus.

Por otro lado, con el objetivo de reducir el número de peticiones que se realizan a memoria principal, la mayoría de procesadores comerciales implementan jerarquías de memoria con hasta tres niveles de memorias cache y grandes memorias cache de último nivel (LLC), que ya alcanzan varias decenas de megabytes [KSB+09]. Además, con estas LLC grandes y otros mecanismos como las capacidades multihilo y la prebúsqueda, se consigue *ocultar* en gran medida las altas latencias de acceso a memoria principal existentes entre los procesadores y las memorias DRAM actuales.

Para mejorar la utilización de las caches, muchos procesadores implementan caches compartidas por un subconjunto de núcleos en al menos un nivel de la jerarquía de memoria. Así, procesadores como el Intel Dunnington [KSB+09] o el Power 5 [SKT+05], presentan caches compartidas por varios núcleos en los niveles L2 y L3 de la jerarquía. Por ejemplo, en el procesador Power 5, cada cache de L2 es compartida por dos núcleos y a su vez, cada cache de L3 es compartida por cuatro núcleos. De manera similar, el Intel Dunnignton implementa caches de L2 compartidas por dos núcleos y una cache de L3 compartida por todos los núcleos del procesador. Más aún, la presencia de núcleos multihilo que presentan muchos procesadores actuales provoca que el comportamiento de caches L1 privadas se asemeje al de una cache compartida, pues son varios los hilos de ejecución que deben compartir el ancho de banda disponible para acceder a ella.

En resumen, los procesadores comerciales actuales suelen implementar, en algunos de los niveles de la jerarquía de memoria, caches compartidas por un número creciente de núcleos computacionales e hilos de ejecución. Por ello, la contención en el ancho de banda disponible puede aparecer en cualquier punto de la jerarquía cache y no solo en el acceso a memoria principal. Por tanto, estos puntos de contención potenciales deben ser considerados en el algoritmo de planificación para maximizar las prestaciones del sistema.

Este trabajo presenta tres contribuciones principales. En primer lugar, se describe el uso de contadores de prestaciones y *huge pages* en el diseño de experimentos, permitiendo controlar los accesos que se realizan sobre cada cache, los conjuntos accedidos y la tasa de accesos. En segundo lugar se caracteriza la sensibilidad de las prestaciones de los benchmarks SPEC CPU2006 a la contención en los diferentes puntos de la jerarquía de memoria de un Intel Xeon X3320. Por último, se propone un algoritmo de planificación para procesadores multinúcleo con caches compartidas, que selecciona los procesos a ejecutar y los ubica en los núcleos de manera que se maximiza la utilización del ancho de banda disponible en el sistema limitando la contención y maximizando las prestaciones.

Los algoritmos de planificación propuestos han sido implementados y se han evaluado en un sistema real con un procesador Intel Xeon X3320 [Int]. Los tiempos de

ejecución de las cargas diseñadas se han comparado con el tiempo de ejecución utilizando la planificación de Linux y un planificador que considera la contención en el acceso a memoria principal [XWY10]. A pesar de que la jerarquía de cache del procesador utilizado en la evaluación de prestaciones es más pequeña que la que presentan algunos procesadores comerciales más recientes, los resultados experimentales muestran que el algoritmo de planificación final ofrece unas prestaciones un 6.55 % superiores a las que presenta el planificador del sistemas operativo Linux, lo que significa duplicar la aceleración alcanzada con una planificación que únicamente considera la contención en el ancho de banda disponible para acceder a memoria principal. Además, dado que se espera que los problemas de contención se incrementen en futuros procesadores, es de esperar que en éstos procesadores, el algoritmo de planificación incremente la mejora de prestaciones respecto la planificación de Linux y la que ofrece un planificador considerando el ancho de banda para acceder a memoria principal.

El resto del trabajo se organiza como sigue. El Capítulo 2 presenta el estado del arte en relación a algoritmos de planificación que consideran la contención en los sistemas. El Capítulo 3 describe el sistema donde se realizarán los experimentos y dos características avanzadas de la arquitectura como son los contadores de prestaciones y las *huge pages*. El Capítulo 4 discute la influencia de la LLC en el rendimiento de los procesadores actuales y presenta una serie de experimentos con los que determinar las características de la LLC y como utilizar *huge pages* para controlar el patrón de accesos de los procesos sobre las caches. El Capítulo 6 explica dos algoritmos de planificación incrementales que consideran la contención a lo largo de toda la jerarquía de cache y la degradación que cada benchmark sufre para realizar la planificación. El Capítulo 7 describe la metodología empleada en los experimentos y discute los resultados obtenidos. Finalmente, el Capítulo 8 presenta las conclusiones del trabajo, las publicaciones relacionadas y el trabajo futuro.

Capítulo 2

Estado del arte

Mucha investigación acerca de la planificación en procesadores multinúcleo se centra en reducir la degradación de prestaciones provocada por la contención en el acceso a memoria principal [ANP03], [ANP04], [XWY10] o a la LLC [QLMP06] [SDR02] [KCS04] [SKE+09] [FBZ10] [ZBF10] [KSCJ10] para maximizar las prestaciones de los CMPs. Además, otros trabajos que consideran la contención de memoria buscan conseguir un compromiso entre energía consumida y tiempo de ejecución.

2.1. Estado del arte relacionado con la contención en el acceso a memoria principal

Con respecto a la contención en el acceso a memoria, Antonopoulos y col. ([ANP03], [ANP04]) proponen diversas estrategias de planificación basadas en el ancho de banda consumido por los procesos ejecutados concurrentemente. En [ANP03], la información acerca del consumo de ancho de banda se obtiene modificando el código fuente de las aplicaciones, mientras que en [ANP04] se presenta una aproximación menos intrusiva basada en utilizar los contadores de prestaciones de los procesadores. En ambos casos, las políticas propuestas tratan de planificar los procesos de forma que el ancho de banda consumido por estos se acerque tanto como sea posible al ancho de banda máximo del bus. En otras palabras, las políticas tratan de planificar los procesos para obtener la utilización óptima del bus, sin infrautilizarlo por debajo de su capacidad para no perder prestaciones pero tratando de tampoco superarla y entrar en contención. En un trabajo posterior [KK06], Koukis y col. se centran planificación de tareas en clústers SMPs. Para ello, además de considerar el ancho de banda consumido por los procesos en el acceso a memoria principal, consideran el ancho de banda de la red requerido por las aplicaciones. La política de planificación que presentan trata de optimizar la utilización de ambos anchos de banda para evitar infrautilizar los recursos o provocar contención.

En un trabajo más reciente, Xu y col. [XWY10] demuestran que los patrones de acceso a memoria irregulares pueden provocar fluctuaciones en la demanda de ancho de banda. Esto significa que si se planifica para alcanzar el ancho de banda máximo del bus, éste puede superarse en algunos intervalos, creando contención. Para resolver este problema proponen utilizar el ancho de banda medio requerido por las aplicaciones para realizar la planificación en lugar de utilizar el ancho de banda pico. Los autores calculan el ancho de banda medio ideal (IABW) de una carga como el número total de accesos a memoria principal dividido por el tiempo de ejecución ideal de la carga. El IABW se ajusta con una regresión polinómica para obtener el IABW óptimo para la planificación. Después, el algoritmo de planificación va seleccionando los procesos de manera que el ancho de banda requerido por los que se ejecutan simultáneamente se aproxime al IABW.

Otros trabajos se centran en diferentes arquitecturas. Por ejemplo en [BZFK10], Blagodurow y col. se centran en arquitecturas NUMA, es decir, sistemas multinúcleos donde existen múltiples dominios de memoria y las latencias de acceso no son uniformes. Los autores discuten que las políticas de planificación del estado del arte que consideran la contención no son efectivas con sistemas NUMA, llegando incluso a empeorar las prestaciones del planificador de Linux, puesto que provocan situaciones donde un hilo se ejecuta en un núcleo mientras su espacio de memoria está ubicado en un dominio diferente. Para resolverlo proponen un algoritmo de planificación que trata de evitar migrar los hilos de ejecución entre dominios distintos, y en caso de que sea necesario migra el espacio de memoria junto con el hilo.

Por otro lado, en [ZJS10] los autores discuten si el rendimiento de las aplicaciones multihilo en CMPs depende de la compartición de la cache, llegando a la conclusión de que su impacto en las prestaciones de la mayoría de aplicaciones es insignificante. No obstante, argumentan que el motivo no es la falta de potencial de la compartición de la cache sino más bien que tanto programadores como compiladores no la aprovechan.

Recientemente en [TMV+11] los autores tratan el impacto de la compartición de los recursos del subsistema de memoria en el rendimiento de aplicaciones de centros de datos. En el estudio se analizan cinco aplicaciones multihilo distintas y sus prestaciones al ejecutarse concurrentemente entre ellas en diferentes situaciones. Contrariamente a lo expuesto en [ZJS10], los autores afirman que en las cargas utilizadas por ellos, la compartición del subsistema de memoria tiene una gran influencia en las prestaciones. Los investigadores estudian cinco aplicaciones diferentes, y llegan a la conclusión que para cada aplicación la mejor ubicación de los threads en núcleos varía en función de las aplicaciones que se ejecutan concurrentemente, y no solo en función de los requisitos de la propia aplicación. Para maximizar las prestaciones, identifican las características relevantes de las aplicaciones y diseñan una política de planificación basada en heurísticas para proporcionar la mejor ubicación de los hilos en núcleos a partir de la

información disponible.

2.2. Estado del arte relacionado con la contención en el acceso a la cache LLC

En relación a la contención por el acceso a la LLC se utilizan dos aproximaciones complementarias: particionado de la cache y planificación considerando la jerarquía de cache. El mecanismo de particionado de la cache [QLMP06] [SDR02] [KCS04] evita la inanición de los procesos que se ejecutan de manera concurrente implementando nuevas métricas hardware y mecanismos para maximizar la productividad y/o la compartición equitativa de la cache. Sin embargo, como señalan Sato y col. en [SKE+09], estos mecanismos pueden reducir seriamente las prestaciones si las aplicaciones concurrentes superan la capacidad de la cache. Por tanto, el algoritmo de planificación debe considerar la geometría de la cache para prevenir este tipo de situaciones.

Por otro lado, diferentes trabajos se centran en la planificación considerando la contención en las caches. En [ENBSH11], Eklov y col. presentan un método con una baja sobrecarga para medir con precisión el rendimiento de las aplicaciones en función del espacio de cache disponible. Para ello, diseñan un microbenchmark que permite configurar el número de vías de la cache que ocupa. El microbenchmark, con diversos hilos de ejecución, es capaz de reservar un espacio de memoria y mantenerlo en la cache realizando accesos continuos sobre los datos, de forma que para las aplicaciones que se ejecutan concurrentemente se simula una situación con un espacio de cache disponible menor.

Centrados en la contención provocada en la cache, Fedorova y col. [FBZ10] [ZBF10] estudian diferentes factores que provocan degradación de prestaciones, entre los que se encuentran la contención en el controlador de memoria, contención en el *front-side bus* (FSB), contención en la cache y contención por la prebúsqueda. Analizando los diferentes factores concluyen que la tasa de fallos en la cache es una buena métrica de la contención, por lo que presentan una política de planificación basada en esta tasa, ya que además se trata de una métrica que se puede obtener fácilmente con los contadores de prestaciones que implementan los procesadores actuales.

A diferencia de los trabajos anteriores, nosotros proponemos una solución global que aborda la contención en el ancho de banda que puede aparecer en cada nivel de la jerarquía de memoria. En la literatura solo hemos encontrado una propuesta similar con un enfoque tan amplio, la de Kaseridis y col. [KSCJ10] pero utiliza algoritmos con particionado de la cache y recursos hardware específicos para evitar la contención. Por el contrario, nuestra propuesta se implementa utilizando los contadores de prestaciones existentes en los procesadores comerciales para controlar el ancho de banda requerido

por cada proceso y por tanto no requiere de ninguna modificación hardware sobre las plataformas existentes.

Capítulo 3

Plataforma experimental

3.1. Descripción del sistema

Como se ha mencionado en el Capítulo 1, los planificadores propuestos se han implementado y evaluado sobre un procesador comercial para obtener sus prestaciones. En concreto, los experimentos se han realizado sobre el procesador Intel Xeon X3320 [Int]. Se trata de un procesador de cuatro núcleos con una jerarquía de cache de dos niveles. Mientras el nivel L1 presenta caches privadas para cada núcleo, el nivel L2 implementa dos caches de 3 MB, cada una compartida por un par de núcleos. Las especificaciones más significativas del sistema se resumen en la Tabla 3.1.

La Figura 3.1 presenta la jerarquía de memoria del Intel Xeon X3320, junto a la del IBM Power 5 [SKT+05]. Los puntos de contención que aparecen en la jerarquía del Xeon X3320 se encuentran en el acceso a memoria principal y en el acceso a cada una de las dos caches de L2, que son compartidas. Si comparamos ésta jerarquía con la que presenta el IBM Power 5, podemos observar que se trata de una jerarquía relativamente pequeña para el tipo de planificación que se propone. Los puntos de contención en el IBM Power 5 son el acceso a cada una de las cuatro caches de L2, el acceso a las dos caches de L3 y el acceso a memoria principal. Es por ello que se podría esperar que las mejoras de prestaciones en un procesador Power 5 fuesen superiores a las que se

Procesador	Intel Xeon X3320
Frecuencia	2.5 GHz
Soporte multihilo	No
Cache L1	Código: 4 x 32 KB Datos: 4 x 32 KB
Cache L2	2 x 3 MB
Memoria	4 GB (2 x 2 GB) DDR2

Tabla 3.1: Especificaciones del sistema

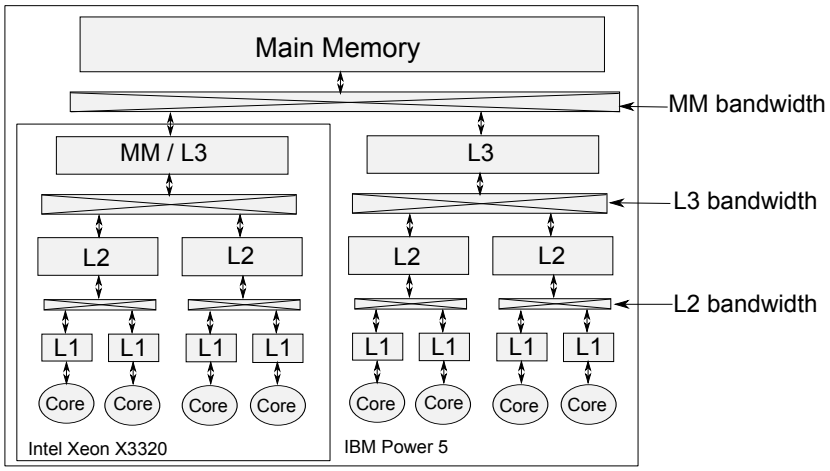


Figura 3.1: Jerarquía de memoria del Intel Xeon X3320 y del IBM Power5

presentan en la evaluación experimental, que son ya de por si buenos, pues se pasa de tres puntos de contención a siete.

El sistema ejecuta una distribución del sistema operativo Fedora Core 10 Linux, con el núcleo 2.6.29, que es el último con soporte para la herramienta de monitorización de los contadores de prestaciones utilizada. El núcleo debe *parchearse* para habilitar la utilización de los contadores de prestaciones.

3.2. Características relevantes de la arquitectura

En esta sección se tratarán los aspectos de la arquitectura de la plataforma experimental relevantes para la obtención de medidas de prestaciones, el estudio de la geometría de las caches y el diseño de los planificadores de prestaciones.

3.2.1. Contadores de prestaciones

La mayoría de procesadores ofrecen un conjunto de registros especiales conocidos como contadores de prestaciones que permiten contabilizar eventos hardware específicos. Cada procesador dispone de su conjunto de eventos hardware que puede monitorizar, algunos de ellos específicos de una arquitectura concreta. Otros, contabilizan eventos hardware típicos como pueden ser los ciclos de ejecución de un proceso, sus instrucciones ejecutadas o el número de accesos y fallos en cada nivel de la jerarquía de cache.

En los planificadores que se diseñan en este trabajo se hace uso de los contadores de prestaciones como una forma eficiente y no intrusiva para medir el ancho de banda que requiere cada proceso de manera dinámica. Otras forma de medir el ancho de banda

requieren instrumentar el código de los programas para poder recoger estos datos. Sin embargo, con los contadores de prestaciones no se requiere ninguna modificación sobre los programas a ejecutar y el sistema operativo puede directamente recoger estos datos mediante la configuración correcta de estos contadores. En este trabajo, los contadores de prestaciones también se han utilizado para caracterizar los benchmarks de la *suite* SPEC CPU2006, estudiar su degradación de prestaciones bajo diferentes condiciones de contención en la jerarquía de memoria (Capítulo 5) y para determinar la geometría de la LLC de la plataforma experimental (Capítulo 4).

Para acceder a los contadores de prestaciones existen diferentes aplicaciones y librerías. En este trabajo se ha optado por utilizar *pfmon* [JJN08] y la librería asociada *libpfm*. La principal característica de ambas utilidades es su capacidad para monitorizar de manera concurrente varios procesos, contabilizando los eventos hardware de manera independiente para cada uno de ellos. El programa *pfmon* se utiliza en la caracterización y evaluación de la degradación de prestaciones de los benchmarks, pues permite acceder a los contadores de prestaciones de manera cómoda, desde la línea de comandos y sin necesidad de modificar los programas. Por su parte, la librería *libpfm* se utiliza en los planificadores implementados para acceder a los contadores de prestaciones de los procesos que se ejecutan.

La Tabla 3.2 indica los eventos que se han medido utilizando los contadores de prestaciones. Aunque el nombre del evento puede variar en función de cada arquitectura, se trata de eventos bastante comunes, por lo que cualquier procesador suele disponer de ellos y por tanto, no consideramos que este requisito sea una limitación para exportar los experimentos realizados o el planificador a la mayoría de procesadores comerciales. El número de instrucciones y ciclos ejecutados por cada proceso se ha utilizado para medir las prestaciones mediante el número de instrucciones por ciclo (IPC) que cada proceso es capaz de ejecutar. Por su parte, el número de fallos en cada estructura de la cache se utiliza para medir el ancho de banda requerido por los procesos en transacciones por microsegundo (BTR) o fallos por cada mil instrucciones (MPKI). El MPKI es una medida más estable del ancho de banda cuando se ejecutan varios procesos de manera concurrente. Al medir los fallos en la cache en función del número de instrucciones, el MPKI no se ve afectado por la variación temporal inducida por la

Evento hardware	Descripción
Unhalted_core_cycles	Ciclos en ejecución del proceso
Instructions_retired	Instrucciones ejecutadas por el proceso
L2_rqsts:self	Fallos del proceso en la cache L1
Last_level_cache_misses	Fallos del proceso en la cache L2 (LLC)

Tabla 3.2: Eventos hardware monitorizados con los contadores de prestaciones

contención. Por tanto, se usará el MPKI allí donde se necesite valorar la contención independientemente de los tiempos de ejecución.

3.2.2. *Huge pages*

La principal dificultad para determinar la geometría de la LLC y diseñar experimentos que la tengan en cuenta para poder realizar el estudio de caracterización es que se trata de una cache físicamente indexada pero virtualmente etiquetada, lo que provoca que accesos a direcciones contiguas en la memoria virtual no sean contiguos en la memoria física. Esto se traduce en que no es posible determinar, desde un programa de usuario, el conjunto de la cache donde un bloque de memoria es almacenado. Para comprender las causas de este problema, empezaremos repasando el concepto de memoria virtual y presentaremos después una solución basada en el uso de *huge pages*.

Traducción de direcciones utilizando memoria virtual

La memoria virtual es una técnica de gestión de memoria utilizada en los sistemas operativos que ofrece dos ventajas principales. En primer lugar, utilizando memoria virtual los programas eliminan la restricción de disponer de un espacio de memoria limitado por la memoria física disponible, de manera que pueden ejecutarse como si dispusieran de una mayor cantidad de memoria. Esto se consigue dividiendo el espacio de direcciones en segmentos llamados páginas, que automáticamente se cargan en memoria física o almacenan en disco en función de si se utilizan o no. En segundo lugar, la memoria virtual ofrece una compartición segura y eficiente de la memoria entre los procesos en ejecución. Dado que solo las páginas que están siendo utilizadas necesitan residir en memoria principal, los programas pueden compartir la memoria disponible de una manera mucho más eficiente.

La Figura 3.2 presenta la traducción de direcciones de las direcciones virtuales utilizadas por las aplicaciones, en direcciones físicas que son las que se utilizan para acceder a la memoria principal y las caches. La dirección virtual se divide en dos campos: número de página virtual y desplazamiento de página. El número de página virtual se forma con los bits más significativos de la dirección. Para las páginas de 4 KB utilizadas típicamente en los sistemas operativos, el desplazamiento de página se forma con los 12 bits menos significativos dejando los 20 bits restantes para identificar la página. Como se muestra en la figura, la TLB traduce el número de página virtual al número de página física asignado, mientras que los bits de desplazamiento se mantienen para la página física. A su vez, la dirección física se divide en tres campos para acceder a la cache: etiqueta de dirección física, índice de cache y desplazamiento de bloque. El ejemplo de la figura corresponde con una LLC de 4 MB, formada por 4096 conjuntos y 16 vías, con una longitud de línea de 64 bytes.

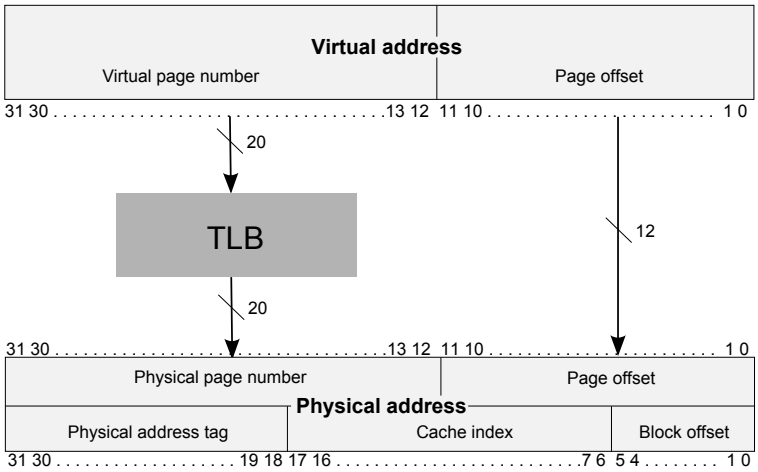


Figura 3.2: Traducción de direcciones para una LLC de 4 MB utilizando páginas de 4 KB

En la figura se puede observar como, asumiendo una longitud de línea de 64 bytes, el índice de cache para caches con más de 2^6 conjuntos (que es el caso de las LLCs) se forma con bits procedentes tanto del desplazamiento de página física como de su identificador. Dado que el número de página física que se asigna a cada página virtual es gestionado por el sistema operativo, no es posible determinar *a priori* a que conjunto de la cache accede una petición a memoria. Es más, por motivos de seguridad (entre otros), Linux lleva a cabo la gestión de las páginas físicas y su asignación a páginas virtuales de una manera pseudoaleatoria. Además, asigna las páginas físicas libres a las páginas virtuales que requieren cargarse en memoria principal y libera las que no se están utilizando, de manera que nada garantiza que dos páginas virtuales adyacentes se carguen en páginas físicas también adyacentes y mucho menos que una página virtual se cargue siempre en la misma página física.

Esto dificulta el diseño de experimentos donde el patrón de accesos a la caches y su ocupación puedan ser controlados. Dada la aleatoriedad del conjunto donde va a mapear un acceso concreto, no es posible determinar si éste se resolverá como un fallo y deberá acceder a memoria principal o si por el contrario se resolverá como un acierto.

Traducción de direcciones con *huge pages*

Para resolver el problema descrito utilizamos *huge pages* en lugar de las páginas estándar de 4 KB que se utilizan habitualmente en los sistemas operativos. Las *huge pages* se propusieron como un mecanismo para incrementar las prestaciones de los sistemas. El mayor beneficio que ofrecen es la reducción de la sobrecarga de la gestión de la memoria virtual. Dado que el tamaño de las páginas virtuales es mayor, el número

de páginas utilizadas por cada proceso disminuye. De este modo, un menor número de entradas en la TLB puede mapear un espacio de direccionamiento virtual mayor, reduciendo también la tasa de fallos en la TLB.

Las arquitecturas modernas dan soporte a las *huge pages* para permitir al sistema operativo utilizar páginas virtuales de mayor tamaño, soportando actualmente una gran variedad de tamaños de página. Por ejemplo, la arquitectura IA-64 soporta páginas desde 4KB hasta 256MB [HMR+00].

Como se discute en algunos artículos [WW09], el uso de *huge pages* se está haciendo más y más necesario a medida que los procesos aumentan sus requisitos de memoria. Sin embargo, las *huge pages* aún no son utilizadas habitualmente, excepto en grandes servidores o clusters que ejecutan aplicaciones de cómputo de altas prestaciones o grandes sistemas de bases de datos.

Linux da soporte a *huge pages* desde el núcleo 2.6 a través del sistema de ficheros *hugetlbfs*. Asumiendo que el soporte para este sistema de ficheros está habilitado en el núcleo, aún se requieren una serie de configuraciones adicionales para su uso. Primero se debe montar el sistema de ficheros *hugetlbfs* y después se debe configurar el número de *huge pages* que el sistema operativo debe reservar, lo cual puede hacerse en el fichero /proc/sys/vm/nr_hugepages. Además, los procesos que deseen utilizar *huge pages* deben reservar memoria directamente sobre el sistema de ficheros *hugetlbfs* con la llamada al sistema mmap().

La Figura 3.3 presenta la traducción de direcciones cuando la memoria de un proceso se mapea utilizando *huge pages* de 2 MB. Las direcciones virtuales se dividen en dos campos, igual que se hace con las páginas de 4 KB, pero en este caso la longitud de los campos es diferente. Dado que utilizando *huge pages* el tamaño de página es de 2

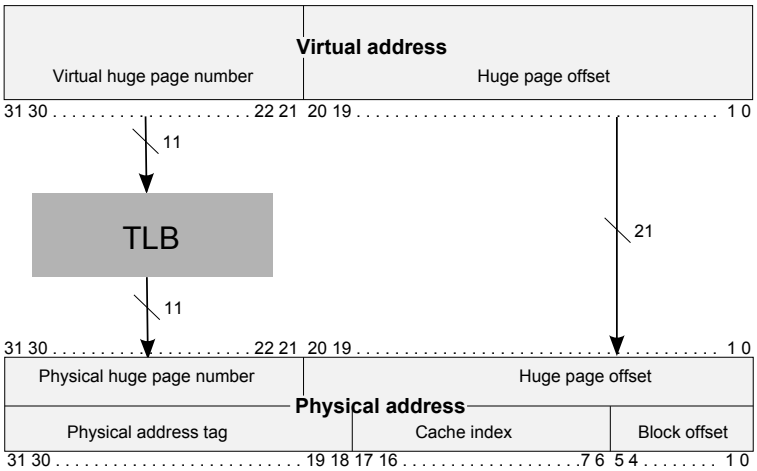


Figura 3.3: Traducción de direcciones para una LLC de 4 MB utilizando *huge pages* de 2 MB

MB, los 21 bits menos significativos de la dirección virtual definen el desplazamiento en la *huge page*, mientras que los 11 bits restantes identifican el número de la *huge page* virtual. La mayor diferencia respecto al uso de páginas de 4 KB es que en este caso, el índice de cache se obtiene directamente de los bits del campo desplazamiento de la *huge page*. De este modo, aunque la *huge page* física se asigne de manera aleatoria a partir de la dirección virtual, se puede determinar en que conjunto de la LLC se mapearán los datos.

Por tanto, el uso de *huge pages* en los experimentos nos va a permitir determinar el conjunto en el que mapea cada acceso a memoria, por lo que el diseño de experimentos puede tener en consideración aspectos como el patrón de accesos a la LLC o la ocupación que los procesos realizan de ella, que típicamente no podían considerarse y que tienen una gran influencia en las prestaciones de la LLC y el sistema.

Capítulo 4

Características de la LLC: geometría y arquitectura

Como se ha avanzado en el Capitulo 1, la jerarquía de cache tiene un papel muy importante en el rendimiento global de los CMPs actuales. La jerarquía de memoria, y en particular la LLC, representan el mayor cuello de botella en los CMPs actuales, con el añadido de que el aumento del número de núcleos e hilos de ejecución en los procesadores, lo cual es la tendencia actual, agravan la situación.

La arquitectura de las caches puede ser privada o compartida, dependiendo de si es accedida por un solo núcleo o concurrentemente por un conjunto de núcleos, y habitualmente se distribuyen en una jerarquía de dos o tres niveles. Para acceder a un bloque de datos, éste se busca en los diferentes niveles de la jerarquía de cache y, si no se encuentra, se accede a memoria principal para recuperarlo. Si finalmente el acceso llega a memoria principal, se puede requerir un tiempo próximo a cien ciclos para obtener el dato. Como ya se ha indicado en capítulos anteriores, para evitar la pérdida de prestaciones provocada por esta espera, los procesadores actuales suelen incluir grandes LLCs, que normalmente se encuentran compartidas por un subconjunto de núcleos para mejorar su utilización.

El rendimiento de las cargas de trabajo depende en gran medida de las prestaciones de la LLC, las cuales dependen a su vez, de los requisitos que cada proceso tenga de la cache y las interferencias que pueden crearse por compartir tanto el ancho de banda para acceder a ella como su capacidad. Es por ello que una política de planificación que busque alcanzar las máximas prestaciones debe considerar el comportamiento de las cargas en función del ancho de banda requerido por cada proceso y la ocupación que presenta en la LLC de cara a asignar los procesos a los núcleos de manera óptima para las prestaciones.

Desafortunadamente, la geometría de las LLC (número de conjuntos, número de vías y tamaño de línea) y el tipo de arquitectura (privada o compartida, y en este caso

como se comparte) son datos habitualmente desconocidos o incompletos. Es por ello que la metodología explicada en este capítulo está orientada a determinar la geometría y arquitectura de las LLC, a partir de tres pequeños tests que determinan de manera experimental estas características. Los algoritmos propuestos se basan en el uso de contadores de prestaciones y *huge pages*.

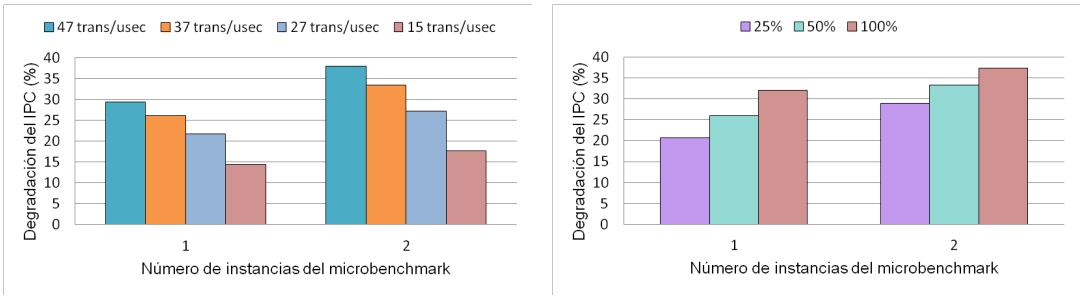
El resto del capítulo se organiza como sigue. En la Sección 4.1 se introduce una series de experimentos que muestran la importancia de de geometría y arquitectura de la cache en las prestaciones del sistema. En la Sección 4.2 se describe el trabajo previo para determinar algunas características de las caches y se discute porque las soluciones propuestas en la bibliografía no se adaptan a nuestros requisitos. La Sección 4.3 presenta los experimentos propuestos para determinar las características de la LLC y los resultados obtenidos en nuestra máquina experimental.

4.1. Motivación

El diseño de microbenchmarks adecuados es una cuestión clave para analizar correctamente la degradación de las prestaciones de los benchmarks en diferentes contextos. Los microbenchmarks son programas que se ejecutan de manera concurrente con las aplicaciones y presentan un comportamiento variable, de manera que se pueden configurar el ancho de banda o la ocupación de la cache para simular los contextos de carga del sistema deseados.

Aunque la geometría de la LLC sea desconocida parece simple diseñar un microbenchmark que provoque contención en el acceso a memoria principal. Esto se puede conseguir reservando una matriz muy grande en memoria y realizando muchos accesos sobre ella, de forma que cuando se repita un acceso a un bloque, este ya haya sido reemplazado de la cache. Para variar la contención introducida, el microbenchmark puede incluir un parámetro que indique el número de instrucciones *nop* entre dos accesos. Sin embargo, un microbenchmark más apropiado controlaría el patrón de accesos que el microbenchmark realiza sobre la cache y por tanto la ocupación que realiza de ella.

La Figura 4.1 muestra el mismo experimento ejecutado con las dos versiones del microbenchmark. La Figura 4.1(a) presenta la degradación del IPC del benchmark *bzip2* al ejecutarse con una y dos instancias del primer microbenchmark variando el número de *nops* para que las instancias presenten, en solitario, BTRs de 47, 37, 27 y 15 transacciones por microsegundo. Por su parte, la Figura 4.1(b) presenta la degradación del IPC de *bzip2* al ejecutarse con una y dos instancias del segundo microbenchmark, con un BTR en solitario para cada instancia de 37 transacciones por microsegundo y variando el número de conjuntos de la cache accedidos. Se puede observar la gran influencia que tiene el número de conjuntos de la LLC accedidos en las prestaciones.



(a) Degradación variando el BTR_{MM} del microbenchmark

(b) Experimento variando la ocupación del microbenchmark

Figura 4.1: Degradación del IPC de *bzip2*

Así, en función de la ocupación de la cache que se configura en el segundo experimento, la degradación del IPC para un microbenchmark con un BTR de 37 transacciones por microsegundo puede superar la que se obtiene con un BTR de 47 o no llegar a la que provoca un BTR de 27.

La influencia de la arquitectura de la LLC tiene un impacto en las prestaciones mayor aún. La Figura 4.2 presenta la degradación de prestaciones que sufre *bzip2* al ejecutarse concurrentemente con una instancia del segundo microbenchmark, con una ocupación de la cache del 50 % de los conjuntos y un BTR variable. Si los dos procesos se ejecutan en núcleos que no comparten la LLC, se observa una degradación de prestaciones máxima alrededor del 5 %. Por el contrario, si los dos procesos se ejecutan en núcleos que comparten la LLC, la degradación de prestaciones varía entre el 15 % y el 30 % en función del BTR del microbenchmark.

A partir de los dos pequeños experimentos presentados, queda clara la influencia que tiene la LLC en las prestaciones de los procesos. Por tanto, es importante conocer la geometría y arquitectura de la LLC, de cara a diseñar experimentos apropiados que simulen correctamente las situaciones de carga que se desea estudiar.

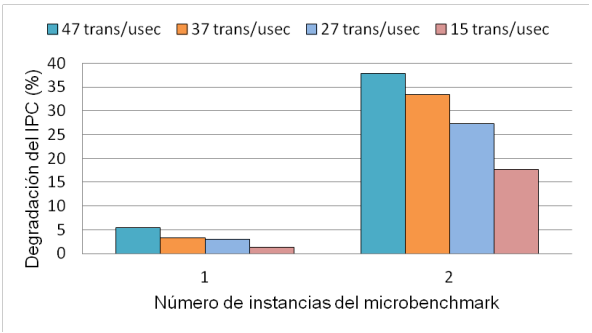


Figura 4.2: Degradación del IPC de *bzip2* variando la ubicación del microbenchmark y su BTR_{MM}

4.2. Trabajo previo

Existe una gran cantidad de trabajos orientados a determinar los parámetros de la jerarquía de memoria, pero ninguno de ellos se adapta completamente a las características de los procesadores y jerarquías de memoria actuales. Un estudio muy conocido para determinar las características de la memoria cache es el que propuso Saavedra [SB92]. Para obtener las características de las caches, Saavedra propone el uso de un benchmark que accede regularmente a los componentes de una matriz grande variando el *stride* entre los accesos. Para cada acceso se mide la latencia con el objetivo de calcular el tiempo de acceso medio durante una larga secuencia de accesos. La información obtenida debe ser analizada después para deducir las características que presenta la jerarquía de cache. El análisis de esta información es mucho más complejo en los procesadores actuales por las diferentes características avanzadas que incorporan. Por ejemplo, el uso de técnicas de prebúsqueda o políticas de reemplazo pseudo-LRU, provocan que algunos accesos que en principio debieran resolverse como fallos en una arquitectura menos avanzada, puedan provocar aciertos en la cache, desvirtuando las medidas temporales y complicando su posterior análisis.

Además de esto, en el trabajo se toma como hipótesis que las caches son virtualmente direccionables, lo que quiere decir que una matriz que ocupe una región contigua de memoria virtual también ocupará una región contigua en la memoria cache. Sin embargo, las arquitecturas de caches actuales son virtualmente indexadas pero físicamente etiquetadas, por lo que las páginas virtuales adyacentes no son necesariamente adyacentes en la memoria física. Es por ello que los algoritmos propuestos no pueden ser aplicados directamente para determinar las características de las caches actuales.

Por otro lado, existen algunas herramientas [MS] [Man04] que son capaces de medir algunos parámetros hardware como la capacidad de la cache, el tamaño de bloque o la latencia. Sin embargo, no ofrecen información acerca de la jerarquía de memoria como la asociatividad de la cache, el número de caches en cada nivel o como estas son compartidas.

Recientemente, en [GDTF+10] González-Domínguez y col. proponen un conjunto de benchmarks para detectar parámetros hardware con una alta influencia en las prestaciones de sistemas de clústers multinúcleo. Además de detectar las caches compartidas, también calculan sobrecargas de memoria y comunicaciones. Como en trabajos previos, las estimaciones se realizan a partir de la medición de tiempos al acceder a diferentes matrices y también se obvian parámetros importantes como la asociatividad o el número de conjuntos de las caches.

En contraposición a estas propuestas, nosotros proponemos una serie de experimentos que permiten determinar completamente las características de las caches: geometría de la LLC (número de conjuntos, número de vías y tamaño de la línea) y su arquitec-

tura (número de caches del último nivel y compartición de las caches). Además, esto se realiza monitorizando los contadores de prestaciones y calculando la tasa de acierto en la LLC, lo cual es una aproximación mucho más directa y simple, pues no requiere el análisis de los parámetros temporales, que además pueden tener interferencias por aspectos como la política de reemplazo de la cache o el *prefetch* hardware.

4.3. Diseño de los tests experimentales

En esta sección se presenta una serie de experimentos que permiten establecer las características de la cache, es decir, su geometría (número de conjuntos, número de vías y tamaño de línea) y el tipo de arquitectura (privada o compartida, y en este caso qué núcleos comparten cada cache). Se diseñan tres experimentos basados en el uso de *huge pages* para controlar el conjunto de la cache al que mapea cada acceso a memoria, y contadores de prestaciones. El primer experimento propuesto determina el tamaño de la línea de cache. El segundo experimento establece el número de conjuntos y la asociatividad de la LLC. Finalmente el tercer experimento identifica el número de caches que forman el último nivel de la cache y, en el caso de que haya varias qué núcleos comparten cada una de ellas.

En la implementación de los pseudocódigos de cada experimento nos aseguramos que los accesos a la cache únicamente se producen por escrituras en el vector de datos. Para ello, los índices de los bucles se mapean en registros y se utiliza el flag de optimización -O1 de gcc para evitar que el compilador optimice el código y pueda alterar los accesos a la cache.

4.3.1. Obtención del tamaño de la línea de cache

Los procesadores actuales utilizan típicamente una longitud de línea de 64 bytes. Sin embargo, esta información es habitualmente ignorada en las especificaciones del procesador. El Algoritmo 4.3 presenta el pseudocódigo diseñado para determinar el tamaño de la línea de cache.

Entrada: LONGITUD_LÍNEA_MATRIZ
$N = 2 * \frac{TAMAÑO\_CACHE}{LONGITUD\_LÍNEA\_MATRIZ}$
char A[N][LONGITUD_LÍNEA_MATRIZ]
for (r=0; r<REPS; r++) **do**
  for (i=0; i<N; i++) **do**
    A[i][0] = 1
  end for
end for

Figura 4.3: Pseudocódigo para determinar la longitud de la línea de cache

El funcionamiento del algoritmo es el siguiente. Para el tamaño de línea de matriz que recibe el algoritmo como parámetro de entrada, se reserva una matriz con un número de líneas, del tamaño indicado, lo suficientemente grande como para la matriz doble el tamaño de la cache. Aunque el tamaño de la cache pueda ser desconocido en este momento, es suficiente con reservar una matriz muy grande, de manera que se asegure que no cabe en la cache. De este modo, cabe esperar que todos los bloques accedidos y guardados en la cache se reemplacen antes de repetirse un acceso sobre ellos aunque la política de reemplazo no sea LRU. Cabe resaltar que las páginas deben ubicarse utilizando *huge pages* dado que, como se ha explicado anteriormente, en caso contrario no se garantiza que los bloques accedidos secuencialmente se almacenen en conjuntos contiguos. Una vez reservada la matriz, el algoritmo entra en un bucle donde el primer byte de cada fila de la matriz se escribe secuencialmente.

El comportamiento esperado es que cuando la longitud de la línea de la matriz coincida con el tamaño de la línea de cache, todos los accesos se mapeen en los conjuntos de manera secuencial. Dado que la matriz dobla la capacidad de la cache y toda la matriz se accede entre dos accesos a la misma posición, un bloque siempre será reemplazado de la cache antes de volver a ser accedido. En esta situación, cuando la longitud de la línea de la matriz y el tamaño de la línea de la cache coincidan, la tasa de acierto en la LLC se aproximará a cero.

Longitudes de la línea de la matriz menores que la longitud de la línea de la cache, aumentarán la tasa de acierto en la cache, pues algunos accesos podrán servirse directamente desde la LLC. Por ejemplo, cuando la longitud de línea sea la mitad que el de la cache, cada acceso a memoria traerá a la cache dos líneas de la matriz, con lo que la tasa de acierto se aproximará al 50 %. Por el contrario, aumentar la longitud de la línea de la matriz mantendrá la tasa de aciertos sobre el 0 % siempre que el número de

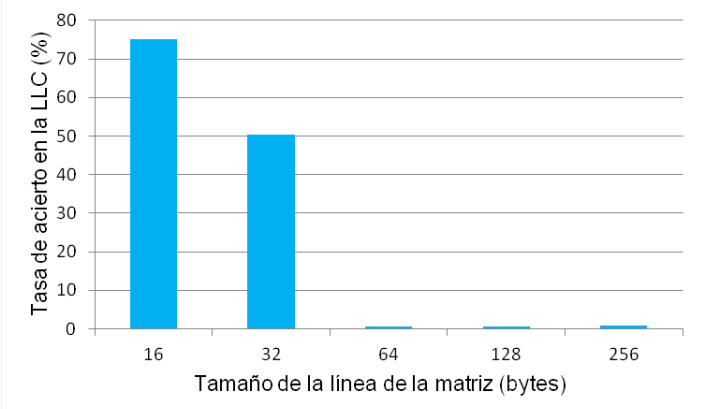


Figura 4.4: Tasa de acierto global en la cache variando la longitud de la línea de la matriz

líneas diferentes sea suficiente como para que se reemplacen en la cache antes de volver a ser accedidas.

La Figura 4.4 presenta los resultados de este experimento en el procesar Xeon X3320 descrito en el Capítulo 3. De los resultados se puede observar que, como se presuponía, el tamaño de la línea de la cache en el procesador utilizado es de 64 bytes.

4.3.2. Obtención del número de conjuntos y vías de la cache

Una vez identificado el tamaño de la línea de la cache, es necesario establecer el número de conjuntos y vías de la cache para determinar su geometría. El número de conjuntos debe ser necesariamente potencia de dos, pues como se ha visto en la Sección 3.2.2 se indexa por un subconjunto de los bits de la dirección física. Además, el tamaño de la cache debe coincidir con el resultado de: #conjuntos * #vías * tamaño de la línea.

La Figura 4.5 presenta el pseudocódigo del algoritmo utilizado para determinar el número de conjuntos y vías de la cache. La idea central del algoritmo es comprobar la tasa de acierto en la cache mientras se varía el número de líneas accedidas. La principal diferencia con el Algoritmo 4.3 es que en este caso los accesos se realizan con *strides* mayores que una línea de cache, en lugar de realizarlos sobre las líneas de cache de manera secuencial. Los parámetros de entrada para el algoritmo son el *stride* con el que se accede a la matriz, que se utilizará para determinar el número de conjuntos de la cache, y el número de accesos sobre posiciones diferentes en la matriz, que determinará la asociatividad de la cache.

El funcionamiento del algoritmo se explica a continuación. En primer lugar se procede a reservar la matriz utilizando *huge pages* como en el experimento anterior. La matriz se reserva con un número de filas igual al número de accesos sobre posiciones diferentes multiplicado por el número de conjuntos, y con una longitud de línea del tamaño de la línea de cache determinada anteriormente. Utilizando el *stride* apropiado (igual al número de conjuntos de la cache) todos los accesos se realizarán sobre el mismo conjunto, empezando a provocar fallos en la cache cuando el número de accesos diferentes supere el número de vías de la cache.

```
Entrada: STRIDE y ACCESOS_DIFERENTES,
  N = ACCESOS_DIFERENTES * STRIDE
  char A[N][LONGITUD_LÍNEA_CACHE]
  for (r=0; r<REPS; r++) do
    for (i=0; i<ACCESOS_DIFERENTES; i+=STRIDE) do
      A[i][0] = 1
    end for
  end for
```

Figura 4.5: Pseudocódigo para determinar el número de conjuntos y vías de la cache

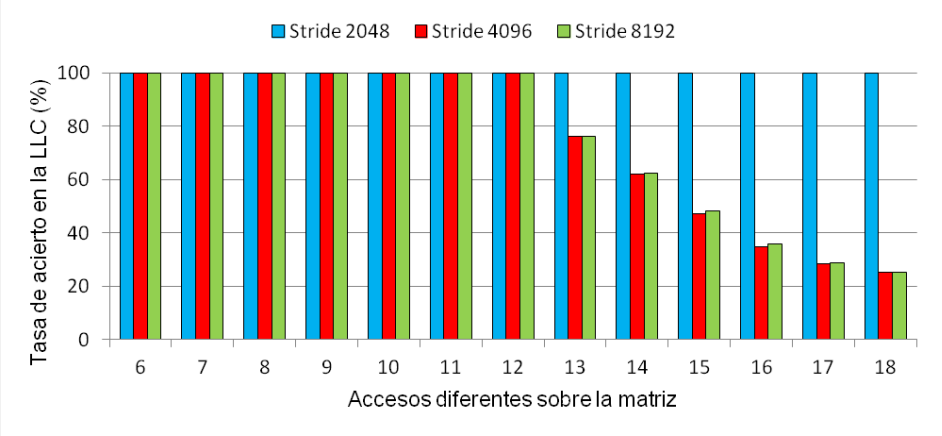


Figura 4.6: Tasa de acierto global en la cache variando el número de accesos diferentes y el *stride*

El algoritmo se ejecuta variando el número de conjuntos y vías tentativamente. Cada ejecución se monitoriza utilizando los contadores de prestaciones para calcular la tasa de acierto global en la cache. El comportamiento esperado es que la tasa de acierto en la cache empiece a disminuir cuando el número de accesos diferentes supere la asociatividad de la cache. De este modo se identifica el número de vías de la cache. Por otro lado, el número de conjuntos de la cache corresponde con el menor *stride* con el que la tasa de acierto en la cache empieza a disminuir.

La Figura 4.6 presenta los resultados del experimento. Se puede observar claramente que la tasa de acierto sobre la cache empieza a disminuir a partir de 12 accesos diferentes sobre la matriz. A partir de este valor, los bloques accedidos no caben en las vías de la cache y empiezan a reemplazarse. Esto significa que el grado de asociatividad del Xeon X3320 es de 12 vías.

Por su parte, el número de conjuntos puede determinarse a partir del menor *stride* a partir del cual la tasa de acierto en la cache disminuye, en este caso, 4096 conjuntos. Son varios los *strides* con los que la tasa de acierto empieza a disminuir, sin embargo, el *stride* mínimo determina el número de conjuntos. Para *strides* mayores que el número de conjuntos, la tasa de acierto también disminuye pues el *stride* es múltiplo del número de conjuntos, de modo que se garantiza que los accesos se realizan sobre un único conjunto, solo que parte de los bloques que se mapearían en el citado conjunto no serían accedidos. Para *strides* menores la tasa de acierto en la cache nunca puede disminuir pues los accesos se reparten por diversos conjuntos y sin que se llegue nunca a superar el número de vías.

Como resumen, este experimento ha determinado el número de conjuntos de la cache en 4096 y el número de vías en 12, a partir del número de accesos diferentes sobre una matriz y el *stride* para acceder a ella. Junto con el tamaño de la línea de

cache determinado antes, nos permite establecer la geometría completa de la cache.

4.3.3. Obtención del número de caches y su grado de compartición

Para maximizar la utilización de las caches los procesadores actuales suelen implementar uno o varios niveles de la jerarquía de memoria con caches compartidas por un subconjunto de núcleos. Determinar los núcleos que comparten la cache es crítico para planificar correctamente aplicaciones que tengan altos requisitos de memoria.

El siguiente experimento determina el número de LLCs incluidas en el procesador y su grado de compartición por diferentes núcleos. Para ello, se pretende establecer el subconjunto de núcleos que acceden a cada cache. La idea es utilizar dos instancias del algoritmo que llenen un conjunto de la cache con sus bloques. Cada instancia se asocia a un núcleo distinto. En el caso en que los núcleos compartan la cache, la capacidad del conjunto se superará, de modo que los procesos realizarán un gran número de reemplazos en los bloques, disminuyendo la tasa de acierto en la LLC. Por el contrario, si no comparten la cache, los accesos se realizarán sobre estructuras distintas y la tasa de acierto en la LLC se mantendrá próxima al 100 %.

Las instancias del algoritmo utilizan el pseudocódigo presentado en la Figura 4.5. En este caso, los parámetros de entrada deberán coincidir con la geometría de la cache para conseguir llenar un conjunto. El experimento consiste en ejecutar dos instancias del algoritmo variando el par de núcleos donde los procesos se ejecutan. Es decir, para evaluar si el núcleo A comparte la LLC con el núcleo B, ejecutamos una instancia del algoritmo en cada núcleo. Esto se repite para cada par de núcleos en el sistema. El comportamiento esperado es que si el par de núcleos comparte la cache, la tasa de

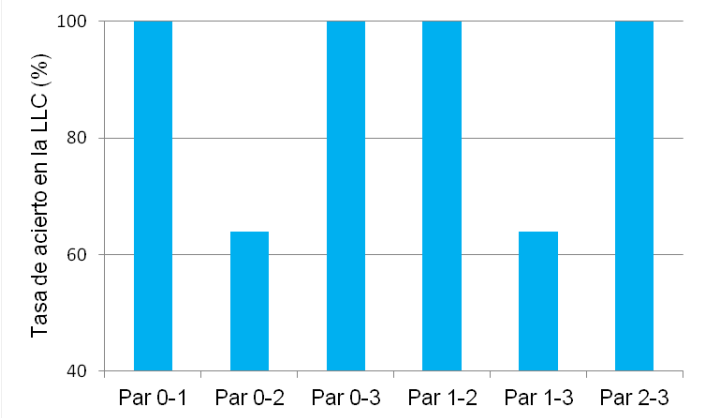


Figura 4.7: Tasa de acierto media en la cache ejecutando dos instancias del algoritmo de llenado de conjuntos en diferentes pares de núcleos

aciertos debe disminuir debido a los reemplazos que las dos instancias deberían estar provocando en el conjunto de la cache que acceden. Por el contrario, si los núcleos no comparten la LLC, los accesos no se afectan entre ellos y la tasa de acierto es del 100 %.

La Figura 4.7 presenta los resultados del experimento. Se observa como los pares de núcleos 0-2 y 1-3 comparten una cache del último nivel, pues la tasa de aciertos en la cache se reduce al 60 % al ejecutar concurrentemente cada uno una instancia del algoritmo. El resto de pares de núcleos no comparte cache, por lo que no hay ninguna interferencia y la tasa de acierto se mantiene próxima al 100 %.

Por tanto, la arquitectura de la LLC que se ha determinado con este experimento es la que se presentó en la Figura 3.1.

Capítulo 5

Análisis de la degradación de prestaciones

Este capítulo explora el comportamiento que presenta el rendimiento de los benchmarks que forman la suite SPEC CPU2006 en la plataforma experimental. Se han utilizado las cargas de entrada de tamaño *train* en la mayoría de benchmarks. Sin embargo, aquellos que presentaban un tiempo de ejecución inferior a los quince segundos, se han analizado con las entradas *ref*, con vistas a utilizarlos posteriormente en el diseño de las cargas para la evaluación de las prestaciones sin que su corta duración pueda ser un inconveniente. En primer lugar se caracterizan los benchmarks midiendo las prestaciones obtenidas cuando se ejecutan en solitario. Después se analiza la degradación de prestaciones provocada por contención en el acceso a memoria principal y a la cache de L2. Para ello se realizan varios experimentos en los que se ejecutan los benchmarks junto con un número y configuración de los microbenchmarks variable, simulando diversas situaciones con consumos de ancho de banda diferentes.

5.1. Caracterización de los benchmarks

Cada benchmark se caracteriza en función de tres índices de prestaciones: instrucciones por ciclo (IPC), transacciones de acceso a L2 por microsegundo (BTR_{L2}) y transacciones de acceso a memoria principal por microsegundo (BTR_{MM}). Para caracterizar los benchmarks se ejecutan en solitario, con el fin de evitar interferencias debidas a la ejecución concurrente con otras aplicaciones, recogiendo en los contadores de prestaciones el valor de los eventos apropiados como se describe en el Capítulo 3.

La Figura 5.1 muestra el IPC para cada uno de los benchmarks, diferenciando los benchmarks de aritmética entera de los de coma flotante. Por su parte, la Figura 5.2 presenta el BTR_{L2} y la Figura 5.3 presenta el BTR_{MM} para cada benchmark, respectivamente. Como se espera, un BTR alto en el acceso a memoria principal provoca

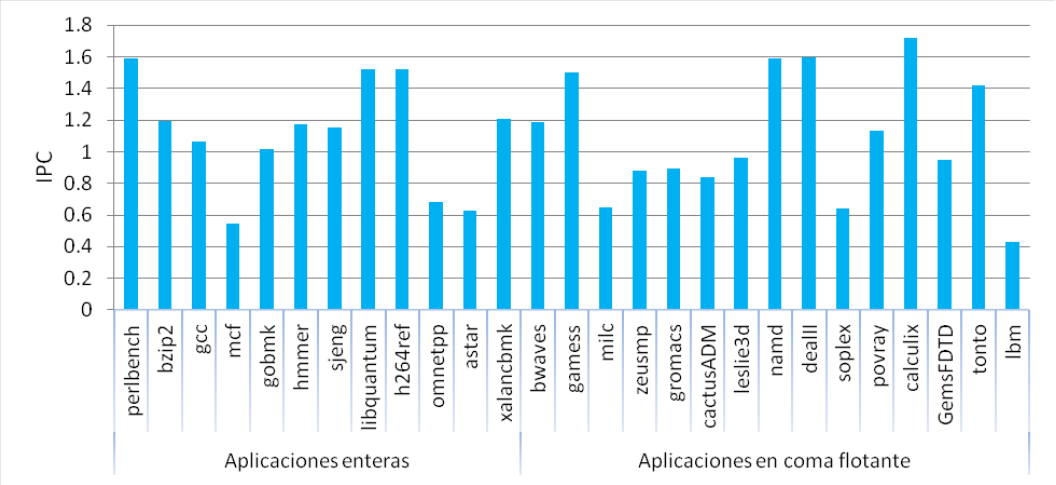


Figura 5.1: IPC para cada benchmark

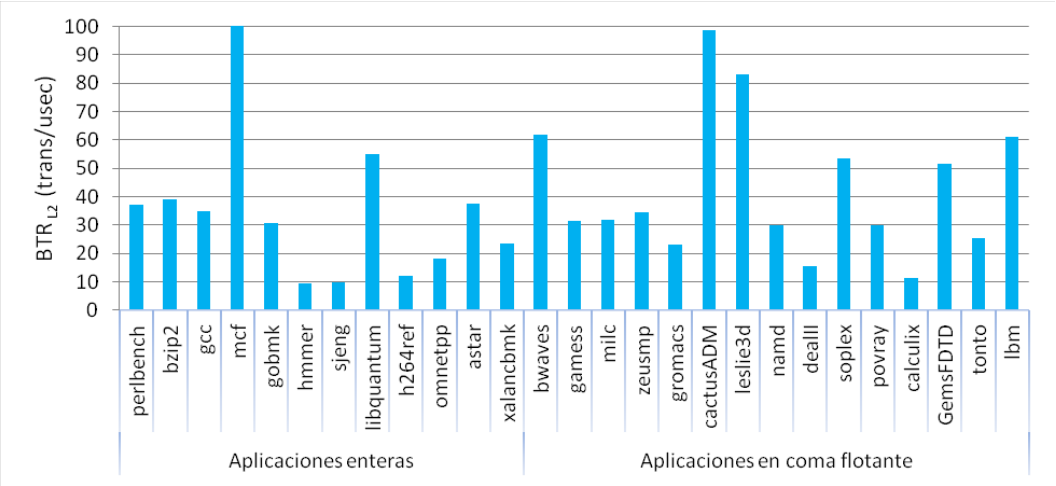


Figura 5.2: BTR_{L2} para cada benchmark

un IPC bajo, debido a los ciclos que el procesador debe esperar para obtener los datos. Este es el caso de benchmarks como *mcf*, *milc* o *lbm*.

Dependiendo del BTR_{MM} de los benchmarks, se pueden clasificar como memory-bounded, cuando sus accesos a memoria principal son lo suficientemente numerosos como para provocar contención en el acceso a memoria. En este caso, el BTR_{MM} del benchmark será alto, y su ejecución concurrente con otros procesos que accedan a memoria provocará contención y afectará al IPC. De manera similar, un benchmark se considerará como L2-bounded, cuando sus accesos a la cache L2 sean muy numerosos, presentando un BTR_{L2} alto y degradando las prestaciones de procesos que accedan frecuentemente a L2 y se ejecuten de manera concurrente.

Es interesante observar que L2-bounded no necesariamente significa memory-bound-

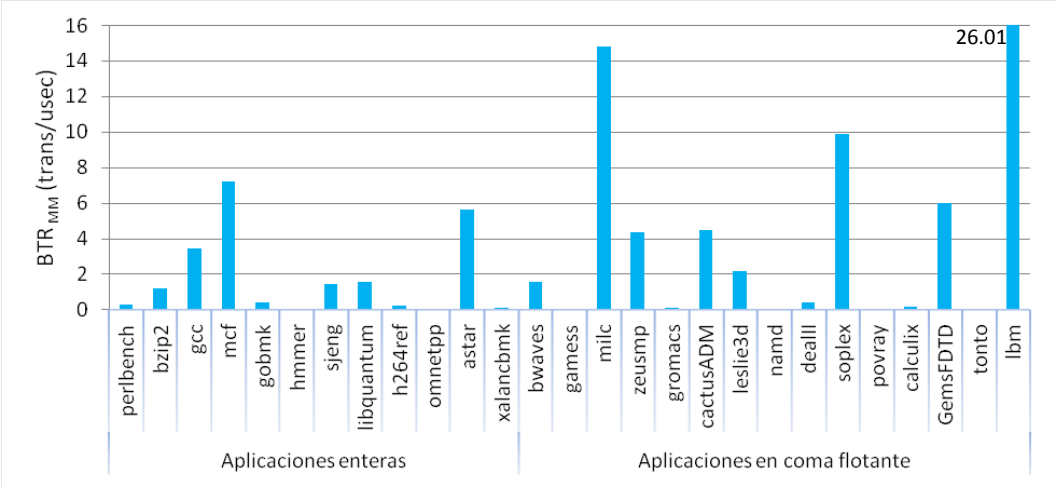


Figura 5.3: BTR_{MM} para cada benchmark

ed, pues aunque el número de accesos a L2 sea elevado, la LLC puede tener un tamaño suficiente para resolver la mayoría de peticiones sin necesidad de acceder a memoria. Tampoco un proceso memory-bounded es necesariamente L2-bounded, pues aunque todos los accesos que se realizan a memoria principal pasan previamente por la L2, el ancho de banda disponible para acceder a ésta es muy superior, con lo que un mayor número de accesos son necesarios para crear contención en este punto. Por ejemplo, *lesli3d* es uno de los benchmarks con un BTR_{L2} mayor, pero esto no se traduce en un BTR_{MM} demasiado alto.

5.2. Diseño del microbenchmark básico

Para analizar la degradación de prestaciones provocada por la contención en el ancho de banda disponible, se diseña un microbenchmark que se ejecutará concurrentemente con los benchmarks SPEC CPU2006, insertando tráfico sintético en el punto de contención deseado. El Algoritmo 5.4 presenta el pseudocódigo del microbenchmark. El funcionamiento del algoritmo es simple y se basa en realizar accesos sobre una matriz reservada, provocando transacciones en la jerarquía de memoria para obtener los datos requeridos. Para el diseño de este algoritmo hemos utilizado las técnicas descritas en el Capítulo 3, lo que nos permite controlar el conjunto de la cache asociado a cada línea de la matriz.

A pesar de su simplicidad, el microbenchmark es altamente configurable. Una característica importante es que permite determinar el nivel de la jerarquía donde ocurren los fallos de cache variando las dimensiones de la matriz reservada. De este modo, si si se producen los fallos en L1, el microbenchmark generará contención en el acceso a

```
Entrada: N, nops y STRIDE
  char A[N][LONGITUD_LÍNEA_CACHE]
  while (1) do
    for (i=0; r<100; i+=STRIDE) do
      A[i][0] = 1;
    end for
    for (i=0; i<#nops; i++) do
      asm("nop");
    end for
  end while
```

Figura 5.4: Pseudocódigo del microbenchmark

L2, convirtiéndose en un co-runner L2-bounded, mientras que si se producen los fallos en la L2, la contención se creará en el acceso a memoria principal y el proceso será memory-bounded. Además de esto, el parámetro de entrada *nops* controla el tiempo que transcurre entre dos accesos sobre la matriz (y por tanto sobre dos transacciones), permitiendo determinar el ancho de banda consumido por el microbenchmark. Por último, variando el *stride* se controla los conjuntos a los que accede el microbenchmark. Por ejemplo, con *stride* uno, el microbenchmark accederá a todos los conjuntos, y con *stride* dos a solo la mitad.

En los experimentos presentados se ha utilizado un *stride* de cuatro, de modo que se accede a la cuarta parte de los conjuntos. Se ha elegido este *stride* por dos motivos. En primer lugar, representa una distancia de 256 bytes entre accesos, suficiente para que la prebúsqueda hardware no sea habilitada para prebúscar los datos. En segundo lugar, este *stride* es apropiado porque permite acceder a un gran número de conjuntos, por lo que el microbenchmark no provoca fallos por conflicto. Se ha medido el impacto en la tasa de fallos de los procesos usando este *stride* y solo se incrementan un 3 % como máximo, al ejecutar un benchmark junto a un microbenchmark. En contraposición, un *stride* que provocara accesos a unos pocos conjuntos produciría un gran número de fallos de conflicto, incrementando colateralmente la tasa de fallos de un benchmark que también usase ese conjunto.

5.3. Degradación debida a contención en el acceso a memoria principal

Para analizar la degradación de prestaciones provocada por contención en el acceso a memoria principal se configura el microbenchmark descrito con un número de líneas de la matriz de manera que cada acceso al vector de datos provoque un fallo en la LLC y por tanto, un acceso a memoria principal. Además el parámetro *nops* se varía para configurar diferentes tasa de acceso a memoria. Para evaluar la degradación, cada

microbenchmark se ejecuta concurrentemente con un número, ubicación en núcleos y BTR_{MM} de los microbenchmarks diferente.

5.3.1. Degradación variando el ancho de banda consumido por los co-runners

El primer experimento se diseña asumiendo que el sistema está completamente cargado, es decir, todos los núcleos se encuentran ejecutando un proceso. Para este fin, se ejecuta cada benchmark junto con tres instancias del microbenchmark. Para analizar la degradación causada por diferentes cantidades de tráfico de acceso a memoria principal, se varía el número de *nops* ejecutados por el microbenchmark. Los valores de BTR_{MM} utilizados para cada instancia del microbenchmark se encuentran entre 70 y 5 transacciones por microsegundo. El primer valor es el máximo obtenido por un único microbenchmark cuando el número de *nops* es cero y cada acceso al vector del microbenchmark se resuelve como un fallo en la LLC. Por otra parte, el BTR_{MM} mínimo es un valor relativamente bajo correspondiente al de un proceso que prácticamente no realiza accesos a memoria principal pero posee un *working set* que no cabe en las memorias cache.

La Figura 5.5 presenta los resultados de este experimento. Se puede observar que el tráfico de acceso a memoria principal generado por los microbenchmarks afecta fuertemente al rendimiento de algunos benchmarks, llegando a degradar las prestaciones de algunos benchmarks por encima de un 60 %. Podemos diferenciar tres grupos de benchmarks en función de la degradación que experimentan. En primer lugar destacamos los benchmarks, que como *lbm* sufren una degradación extrema, superior al 30 %. Se

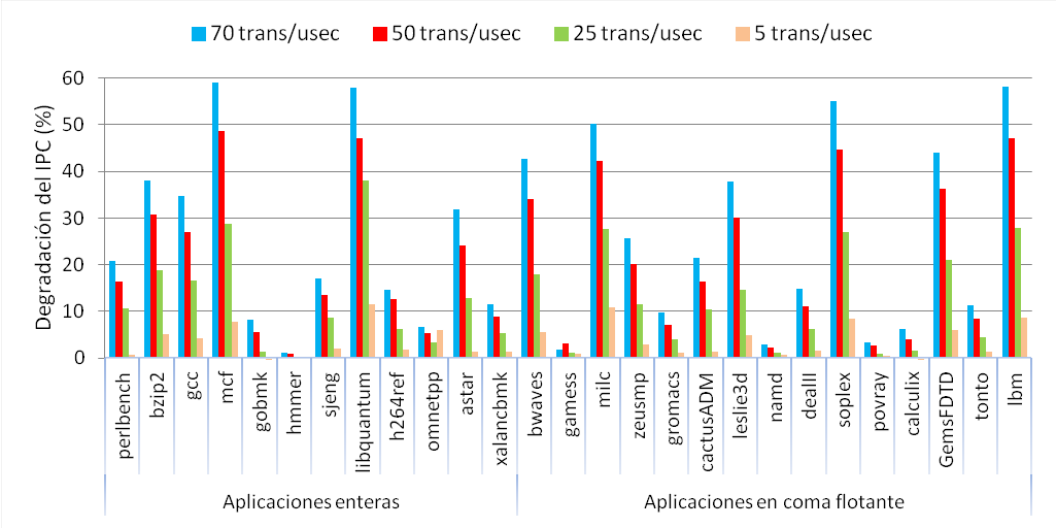


Figura 5.5: Degradación del IPC variando el BTR_{MM} de los co-runners

trata de benchmarks con un BTR_{MM} muy alto, y que por tanto, son muy sensibles a la contención en el acceso a memoria principal. Un segundo grupo incluye los benchmarks que sufren una degradación media, como *cactusADM*, con una degradación entre el 10 % y el 30 %, y que poseen BTR_{MM} intermedios. Por último, tenemos los benchmarks que sufren una baja degradación, inferior al 10 % como *h264ref* o incluso nula como *hmmer*. En este caso son benchmarks con un BTR_{MM} muy bajo y a los que por tanto, el hecho de que el bus esté muy cargado y haya contención para acceder a memoria principal no les afecta. En la figura también se observa que a medida que aumenta el BTR_{MM} de los microbenchmarks la degradación crece. Finalmente, cabe destacar que benchmarks como *milc* o *lbm* presentan degradaciones superiores al 15 % cuando se ejecutan con tres instancias del microbenchmark con un BTR_{MM} mínimo.

5.3.2. Degradación variando el número y ubicación de los co-runners

El segundo experimento analiza la degradación sufrida por los benchmarks al ejecutarse con un número variable de microbenchmarks en diferentes ubicaciones. Cada instancia del microbenchmark es configurada para que presente un BTR_{MM} en solitario de 50 transacciones por microsegundo. La Figura 5.6 presenta los cuatro escenarios analizados que se corresponden con las siguientes situaciones:

- (a) El benchmark se ejecuta con una instancia del microbenchmark pero no comparten la memoria cache de L2.
- (b) El benchmark se ejecuta con una instancia del microbenchmark, en este caso, compartiendo la memoria cache de L2 y el ancho de banda disponible para acceder a ella.
- (c) El benchmark se ejecuta con dos instancias del microbenchmark, compartido su memoria cache de L2 y el ancho de banda disponible para acceder a ella con una de las instancias.
- (d) El benchmark se ejecuta con tres instancias del microbenchmark. En este caso, el benchmark comparte la cache de L2 con una instancia, mientras que las otras dos comparten la otra cache de L2.

Los resultados de este experimento se presentan en la Figura 5.7. Si se compara la degradación de prestaciones sufrida por los benchmarks cuando cada benchmark se ejecuta concurrentemente con una instancia del microbenchmark (escenarios a y b), se observa que la mayoría de benchmarks presentan una degradación mayor o igual al compartir la cache de L2 (escenario b). Esto se debe a que en esta situación, además

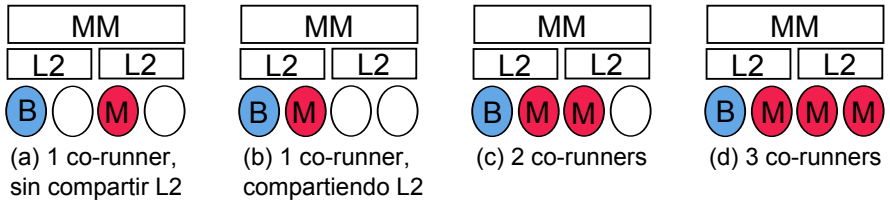


Figura 5.6: Ubicación de los procesos. Leyenda: B (Benchmark) M (Microbenchmark)

de compartir el acceso a la memoria principal, se comparte también el acceso y la capacidad de la cache de L2, por lo que potencialmente el número de conflictos en la cache será mayor y además se puede sufrir contención en el acceso a L2. Al ejecutar el benchmark junto a dos instancias del microbenchmark (escenario c), la degradación se incrementa notoriamente respecto a los escenario a y b, llegando a doblarse en mucho casos. Sin embargo, al incluir una tercera instancia del microbenchmark, el incremento de la degradación de prestaciones se reduce en la mayoría de casos. Esto es debido a que con un benchmark y dos instancias del microbenchmark se consigue saturar el acceso a la memoria principal en la mayoría de los casos.

La degradación de prestaciones observada por contención en el acceso a memoria muestra claramente la necesidad de utilizar algoritmos de planificación que consideren la contención en el acceso a la jerarquía de memoria para seleccionar los procesos que se ejecutarán durante cada *quantum*. Si no se tiene en cuenta esta contención se pueden degradar las prestaciones, llegando a superar el 60 %. Además, con el incremento previsible del número de núcleos e hilos de ejecución en los próximos procesadores la contención será mayor, y con ella, la reducción de prestaciones.

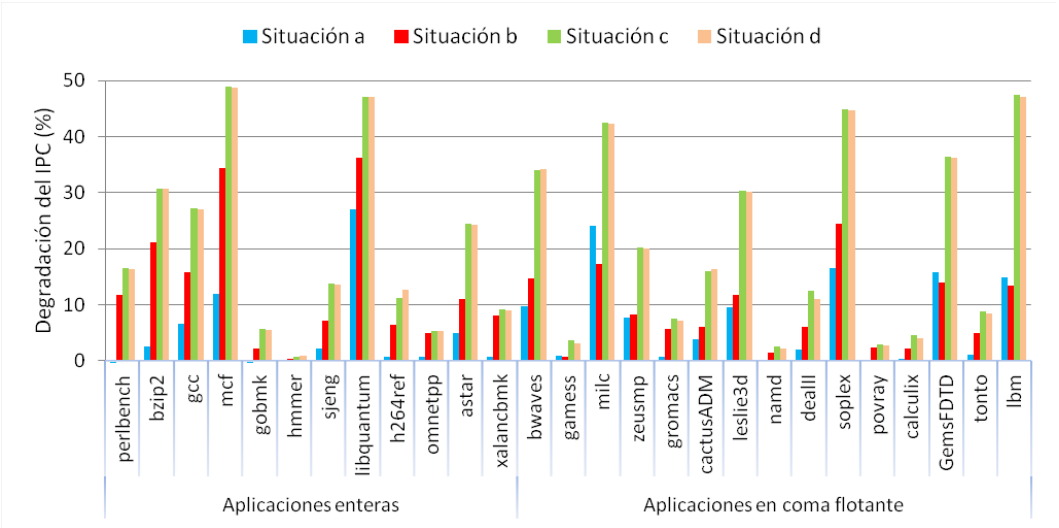


Figura 5.7: Degradación del IPC variando el número y ubicación de los co-runners

5.3.3. Degradación del IPC para un ancho de banda total consumido

El siguiente experimento analiza la degradación del IPC que sufre cada benchmark asumiendo un determinado ancho de banda total consumido por los procesos en ejecución concurrente. Se evalúa la degradación para un consumo total del ancho de banda de 30 transacciones por microsegundo, que es el el IABW medio de las cargas utilizadas para la evaluación de las prestaciones del planificador. El experimento reproduce la situación creada por un planificador que trata de mantener un ancho de banda consumido estable durante la ejecución completa de una carga. Por tanto, el experimento ofrece una aproximación de la degradación de prestaciones que sufrirá cada benchmark cuando se ejecute concurrentemente con un algoritmo de planificación que tenga en cuenta la contención.

Para evaluar la situación descrita, cada benchmark se ejecuta concurrentemente con tres instancias del microbenchmark descrito en la Sección 5.2. La dimensión de la matriz se fija para que los fallos en la jerarquía de cache se produzcan en la LLC y el número de *nops* de las instancias del microbenchmark se configura en función del BTR_{MM} del benchmark analizado para conseguir el BTR_{MM} total deseado. Puesto que para cada benchmark, el BTR_{MM} que puede obtener depende de los procesos que se ejecutan concurrentemente, el número de *nops* debe configurarse de manera independiente para cada benchmark.

La Figura 5.8 presenta los resultados de este experimento. Podemos clasificar los benchmarks en dos clases distintas a partir de su degradación de prestaciones. La

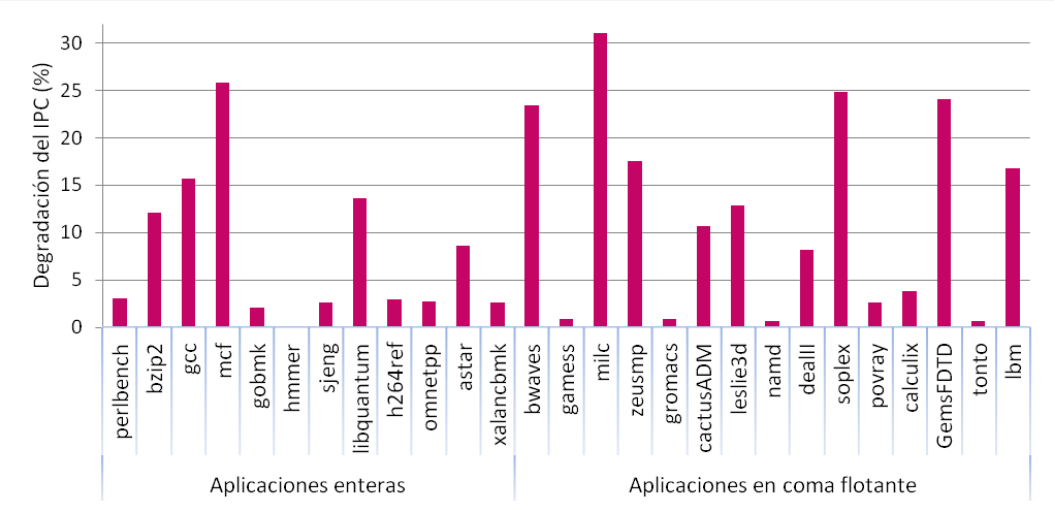


Figura 5.8: Degradación del IPC sufrida por los benchmarks con un BTR_{MM} total de 30 trans/usec

primera clase incluye los benchmarks poco sensibles a la contención en el acceso a memoria principal, dado que el número de accesos que realizan a memoria principal es bajo. Los benchmarks que pertenecen a este grupo presentan una degradación del IPC inferior al 5 %, y si observamos su BTR_{MM} (Figura 5.3) vemos que no alcanza 1 transacción por microsegundo. Por el contrario, la segunda clase incluye los benchmarks con un BTR_{MM} mayor, que sufren también una degradación de prestaciones más alta, que varía entre el 8 % y el 31 %. Sólo dos benchmarks presentan una degradación entre el 5 % y el 10 %. Nótese que la degradación no es lineal con el BTR_{MM}, sino que depende del propio benchmark. Así, el benchmark *lbm* presenta el mayor BTR_{MM} pero no sufre la mayor degradación del IPC.

5.4. Degradación debida a contención en el acceso a la cache L2

Para evaluar la degradación de prestaciones provocada por la contención en el acceso a la cache L2, se diseña un experimento similar al utilizado para analizar la degradación por contención en el acceso a memoria principal. En este caso, únicamente dos núcleos comparten el acceso a cada cache L2, por lo que cada benchmark se ejecutará de manera concurrente con una instancia del microbenchmark. El microbenchmark se configurará para que cada acceso al vector provoque un fallo en la cache L1 y un acierto en la cache L2, evitando de esta forma que posibles fallos en L2 causen contención en el acceso a memoria e interfieran con la contención en el acceso a L2. Tal como hemos hecho en los experimentos previos, se configura el BTR_{L2} del microbenchmark para ofrecer tasas de accesos entre 290 transacciones por microsegundo, que es el máximo cuando el microbenchmark no incluye *nops* y 20, que es un valor suficientemente pequeño como para no crear excesiva contención.

La Figura 5.9 muestra los resultados del experimento descrito. Se puede observar que algunos benchmarks presentan degradaciones importantes, por encima del 7 % al ejecutarse con una única instancia del microbenchmark compartiendo la cache L2. Además, casi la mitad de los benchmarks presentan una degradación que ronda el 5 %. También es interesante observar que benchmarks como *cactusADM*, *povray* o *h264ref* presentan una degradación similar o superior por contención en el acceso a memoria principal que por contención en el acceso a la cache L2, cuando se ejecutan junto a una única instancia del microbenchmark.

Como se esperaba, la degradación de las prestaciones es más alta (salvo excepciones) por la contención en el acceso a memoria principal que por contención en el acceso a L2. Esto es debido a que el acceso a memoria principal se realiza sobre la DRAM, que se encuentra fuera del chip y presenta un ancho de banda menor y una mayor latencia, con

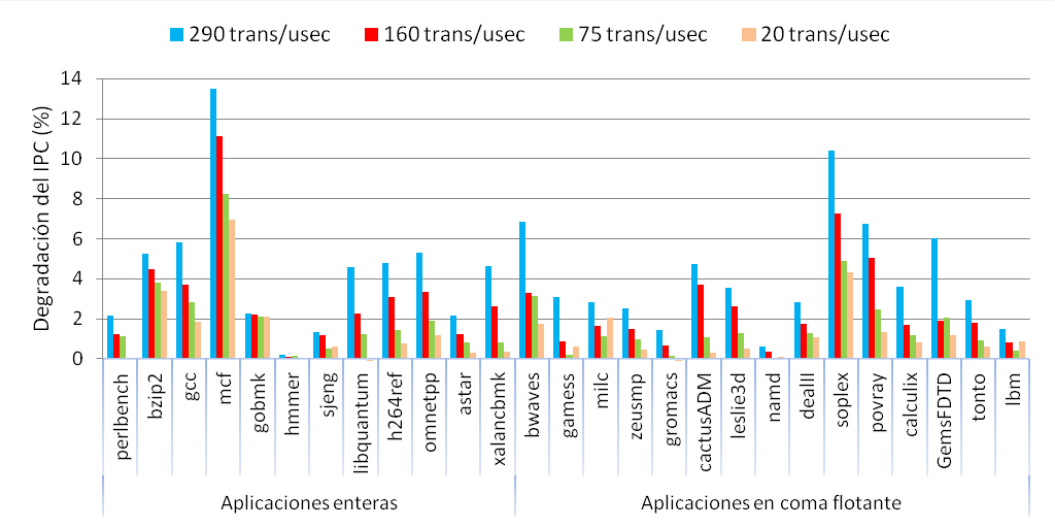


Figura 5.9: Degradación del IPC variando el BTR_{L2} de los co-runners

lo que las peticiones pueden agruparse más fácilmente. Sin embargo, cabe destacar que en el análisis de la degradación por contención en el acceso a la cache L2, únicamente dos procesos realizan peticiones sobre ella. Dado que la tendencia de la industria es a incrementar tanto el número de núcleos computacionales como el número de hilos de ejecución, se espera que en futuros procesadores el número de procesos que realizan peticiones sobre cada cache sea mayor, con lo que también crecerán la contención y la degradación de prestaciones asociada.

Por tanto, en este trabajo se identifica la necesidad de una política de planificación que no solamente considere el ancho de banda requerido por los procesos para acceder a memoria principal, sino que tenga en cuenta el ancho de banda requerido en todos los niveles de la jerarquía de memoria con el fin de reducir la contención global en el sistema y maximizar las prestaciones.

Capítulo 6

Algoritmos de planificación propuestos

La degradación de prestaciones estudiada en el Capítulo 5 muestra la necesidad de un algoritmo de planificación que considere el ancho de banda disponible en cada punto de contención de la jerarquía de memoria del procesador y no solamente en el acceso a memoria principal, para reducir la contención global y maximizar las prestaciones. Los trabajos previos se han centrado en la planificación de procesos considerando la contención en el acceso a memoria principal, por lo que, simplemente seleccionaban los procesos más adecuados para ejecutarse en cada *quantum* atendiendo al ancho de banda disponible para acceder a memoria principal, dejando al sistema operativo la asignación de los procesos a núcleos concretos. Por el contrario, los algoritmos de planificación que proponemos consideran la contención a lo largo de toda la jerarquía de memoria, por lo que, además de seleccionar los procesos a ejecutar, los asigna al núcleo que mejor se adapta a sus requisitos de ancho de banda a lo largo de la jerarquía en función de los procesos que se van a ejecutar de manera concurrente. Cabe destacar también que nuestras propuestas de planificación son soluciones globales que abordan la contención a lo largo de toda la jerarquía de cache, adaptándose automáticamente a la jerarquía que presenta cada procesador sin necesidad de modificar el algoritmo.

6.1. Algoritmo base: planificación considerando el ancho de banda de acceso a memoria principal

Para tratar la contención en el acceso a memoria principal se han propuesto numerosos planificadores que consideran la contención en el acceso a memoria principal. Muchas de las propuestas funcionan de forma similar. Al finalizar un *quantum* bloquean los procesos en ejecución, leen los contadores de prestaciones y actualizan el ancho de

banda requerido por los procesos para el siguiente *quantum*. Después, el planificador selecciona los procesos que se ejecutarán concurrentemente durante el siguiente *quantum* atendiendo al ancho de banda que cada uno espera consumir.

Típicamente, los algoritmos tratan de mantener la máxima utilización del bus, seleccionando los procesos de manera que en cada *quantum* el ancho de banda consumido se aproxime al ancho de banda pico del bus [ANP03] [ANP04]. Sin embargo, trabajos recientes [XWY10] han demostrado que la contención puede existir antes de que el ancho de banda utilizado alcance el ancho de banda pico.

En este trabajo se utiliza como planificador que considera el ancho de banda disponible para acceder a memoria el algoritmo propuesto por Xu y col. en [XWY10]. En dicha propuesta se define un *ancho de banda medio ideal* (IABW) que cuantifica la demanda de ancho de banda media de una carga. La contención en el acceso a memoria principal se reduce planificando los procesos de forma que su ancho de banda consumido en cada *quantum* se aproxime al IABW. De esta forma, las peticiones a memoria se equilibran a lo largo de la ejecución completa de la carga, aumentando sus prestaciones.

6.2. Planificación considerando el ancho de banda de la jerarquía de memoria

El planificador propuesto aborda el ancho de banda en cada punto de contención y planifica los procesos en n pasos (tantos como niveles de cache). Para este fin, el planificador debe monitorizar los fallos que experimenta cada proceso en tiempo de ejecución en cada uno de los niveles de la jerarquía de memoria. La planificación sigue una estrategia top-down, empezando desde la memoria principal, donde se eligen los procesos a ejecutar, hasta el primer nivel de cache, donde cada proceso se asocia a un núcleo concreto. De esta forma, la planificación trata en primer lugar con los puntos de contención que potencialmente tienen un mayor impacto en las prestaciones, que son los que se encuentran más cerca de la memoria principal y donde las latencias y el grado de compartición de las estructuras son mayores. Para poder distribuir los accesos a memoria durante el tiempo de ejecución, el planificador requiere que los procesos que constituyen la carga se envíen junto con su tiempo de ejecución y BTR_{MM} medio cuando se ejecutan en solitario. A partir de estos valores se calcula el IABW de la carga, que corresponde con la suma de accesos a memoria de los procesos que forman una carga, dividido por el tiempo de ejecución ideal de la carga, es decir, el tiempo de ejecución que tendría sin contención. Este valor cuantifica el número de accesos medio de la carga durante su tiempo de ejecución y se utilizará para seleccionar los procesos a ejecutar.

El Algoritmo 1 presenta el pseudocódigo del algoritmo de planificación propuesto. Su funcionamiento se puede dividir en tres partes: i) detención de los procesos en ejecución y actualización de los BTR previstos en cada nivel de la jerarquía de cache, ii) selección de los procesos a ejecutar en función de su ancho de banda de acceso a memoria principal requerido, iii) asignación de los procesos a una cache concreta en cada nivel de la jerarquía. Esta última parte se repite para cada nivel de la jerarquía de cache hasta llegar a la cache L1, donde los procesos se asignan directamente a los núcleos.

Algoritmo 1 Planificador considerando el ancho de banda en la jerarquía de cache

Requiere que los procesos que forman las cargas se envíen con su tiempo de ejecución y BTR_{MM}

$IABW = \frac{\sum_{p=0}^{P}(BTR_{MM}^{p})*T^{p}}{\frac{\sum_{p=0}^{P}T^{p}}{\#n\acute{u}cleos}}$

while haya procesos sin finalizar **do**
  Detiene los procesos en ejecución y los inserta al final de la cola de procesos.
  for cada proceso P ejecutado durante el último *quantum* **do**
    for cada nivel L de la jerarquía con caches compartidas **do**
      Actualiza el BTR de P en el nivel de cache L
    end for
  end for
  $\text{BW}_{Restante}$ = IABW, $\text{CPU}_{Restante}$ = núcleos
  Selecciona el primer proceso de la cola y actualiza $\text{BW}_{Restante}$ y $\text{CPU}_{Restante}$
  while $\text{CPU}_{Restante} > 0$ **do**
    Selecciona el proceso P que maximiza
    $\text{FITNESS(p)} = \frac{1}{\left|\frac{BW_{Restante}}{CPU_{Restante}} - BW_{requerido}^{P}\right|}$
    y actualiza $\text{BW}_{Restante}$ y $\text{CPU}_{Restante}$
  end while
  for cada nivel L de la jerarquía con caches compartidas **do**
    $\text{AVG}_{BTR}\text{(L)} = \frac{\sum BTR\ \ en\ \ L(i-1)}{\#Caches\ \ en\ \ L}$
    for cada cache en el nivel L **do**
      $\text{BW}_{Restante}$ = AVG_{BTR}(L), $\text{CPU}_{Restante}$ = núcleos que comparten la cache L
      while $\text{CPU}_{Restante} > 0$ **do**
        Selecciona el proceso P que maximiza la función FITNESS(p) y actualiza $\text{BW}_{Restante}$ y $\text{CPU}_{Restante}$
      end while
    end for
  end for
  Desbloquea los procesos seleccionados y los asocia al núcleo seleccionado.
  Duerme durante el *quantum*
end while

La primera parte del algoritmo se encarga de detener los procesos en ejecución una vez finaliza el *quantum*. Una vez detenidos, se accede a los contadores de prestaciones donde se recupera la información acerca del número de fallos registrados para cada proceso en cada uno de los niveles de la jerarquía de cache y el número de ciclos que

el proceso ha pasado en ejecución. A partir de estas variables se calculan los BTRs del *quantum* anterior para cada nivel de la jerarquía de la cache. Los BTRs calculados se utilizarán como el BTR estimado para el siguiente *quantum*. Una vez actualizados sus requisitos de ancho de banda, los procesos se insertan al final de la cola de procesos pendientes.

El siguiente paso en la planificación consiste en seleccionar los procesos que se ejecutaran durante el próximo *quantum*. El primer proceso en la cola de procesos pendientes se selecciona automáticamente en cada *quantum* para evitar la inanición de algún proceso si sus requisitos de ancho de banda pueda no lo hacen un candidato adecuado para entrar en ejecución. El resto de procesos hasta completar el número de núcleos se selecciona atendiendo a su BTR_{MM}. Para seleccionarlos se utiliza la función de Fitness, adaptada a partir de la que se presenta en el artículo [XWY10]. La función de Fitness cuantifica, para cada proceso, como se ajusta su BTR_{MM} estimado al ancho de banda restante (BTR_{MM} restante) dividido por el número de núcleos pendientes de asignar. Esto se hace para repartir el ancho de banda disponible entre los núcleos y tratar de evitar, cuando sea posible, que un proceso acapare todo el ancho de banda restante. En cada iteración, que se repite hasta completar el número de procesos a seleccionar, se elige el proceso con un mayor fitness, y se actualiza el número de núcleos disponibles restantes así como el BTR_{MM} restante para seleccionar el siguiente proceso. El resultado de este paso es la lista de procesos seleccionados para ejecutarse en el siguiente *quantum*.

A partir de aquí, el algoritmo de planificación trata con los puntos de contención ubicados en la jerarquía de cache. En cada nivel de la jerarquía que contiene memorias cache compartidas por un subconjunto de núcleos, el algoritmo ubica cada proceso en una cache concreta, de manera que los requisitos de ancho de banda en todas las caches de ese nivel queden equilibrados. Para ello, se calcula en primer lugar el AVG_BTR como la suma del BTR de los procesos seleccionados para ese nivel dividido entre el número de caches compartidas. Para cada una de las caches, se inicializa el BTR restante con el ABG_BTR y se van ubicando los procesos siguiendo la función de Fitness descrita anteriormente.

Como se ha indicado, estos pasos se repiten para cada nivel de la jerarquía de cache con memorias cache compartidas. Nótese que cuando se asignan los procesos en el último nivel de la jerarquía con memorias cache compartidas, los procesos se pueden asociar directamente a los núcleos, pues la ejecución en uno u otro núcleo de entre los que comparten el acceso a la cache es indiferente. Una vez finalizado este proceso, el planificador se encarga de asociar los procesos a los núcleos seleccionados, preparar la monitorización de los contadores de prestaciones y relanzar a ejecución los procesos indicados. Después, el planificador debe esperar a que finalice el *quantum* para volver a iniciar el algoritmo.

6.3. Planificación considerando la degradación del IPC

Para mejorar las prestaciones de la planificación, la información obtenida en la caracterización de prestaciones debería aportar información útil para planificar y ubicar los procesos a ejecutar. Dado que la planificación realizada trata de aproximar el ancho de banda consumido en el acceso a memoria principal durante cada *quantum* al IABW, sería interesante considerar la degradación de prestaciones que cada proceso presenta en esta situación para tratar de favorecer los benchmarks que presenten una degradación mayor. Las cargas utilizadas presentan IABWs entre 20 y 40 transacciones por microsegundo, siendo 30 trans/usec la media. Como se observó en la degradación de prestaciones analizada en la Sección 5.3.3, los diferentes benchmarks presentan diferentes degradaciones de prestaciones en estas condiciones de carga, y si el planificador es capaz de utilizar la información de la caracterización para tomar mejores decisiones, se puede esperar una mejora en las prestaciones.

En la caracterización presentada en la Sección 5.3.3, se diferenciaban los benchmarks en dos clases, en función de la degradación que sufría su IPC. La primer clase incluía los benchmarks que presentan una degradación inferior al 5 %. Podemos considerar estos benchmarks como *poco sensibles* al ancho de banda total consumido por los procesos en ejecución, puesto que la degradación que experimentan es baja. En contraposición, la segunda clase incluía los benchmarks con una degradación de prestaciones entre el 5 % y el 35 %. La degradación de prestaciones de estos benchmarks es alta, por lo que se puede considerar que los benchmarks de este grupo como *sensibles* al ancho de banda total consumido por los procesos en ejecución.

La idea clave de la mejora del algoritmo de planificación que presentamos consiste en favorecer el rendimiento de los benchmarks *sensibles* al ancho de banda total consumido por los procesos en ejecución concurrente. Para ello, cuando un proceso *sensible* se selecciona para ejecutarse durante el siguiente *quantum*, el planificador trata de seleccionar los procesos que se ejecutarán concurrentemente alcanzando un BTR total inferior al IABW calculado para la carga. En contraposición, los benchmarks poco *sensibles* al ancho de banda total consumido, se ejecutarán en situaciones donde el ancho de banda total supere el IABW calculado. Puesto que los benchmarks *sensibles*, que presentan una mayor degradación, se ejecutan en condiciones de ancho de banda más favorables, y los benchmarks *poco sensibles*, con menor degradación, se ejecutan cuando el anchos de banda consumido es mayor, se espera que el resultado ofrezca una mejora en las prestaciones globales de las cargas.

Para incluir esta idea en el algoritmo de planificación se introduce un coeficiente de penalización que favorece o dificulta la selección de cada proceso, en función del ancho

de banda acumulado que prevén consumir los procesos ya seleccionados. El coeficiente de penalización se obtiene como una parte proporcional de la degradación del IPC que presenta cada benchmark cuando el ancho de banda total consumido por los procesos en ejecución concurrente es de 30 transacciones por microsegundo. Concretamente, en la plataforma experimental se considera un proceso de tipo *sensible* si tiene más del 5 % de degradación, en cuyo caso se define el coeficiente de penalización como el 20 % de la degradación de prestaciones. En caso contrario, si el proceso es *poco sensible*, el coeficiente se define como cero. Se evaluaron diferentes coeficientes a partir de la caracterización, siendo esta propuesta la que obtuvo la mayor mejora de prestaciones entre las cargas estudiadas. Nótese también que los coeficientes propuestos no solo diferencian entre benchmarks *sensibles* y *poco sensibles*, sino que además son mayores cuanto mayor es la degradación de los benchmarks, con lo que cuanto mayor sea ésta, más favorable será la situación donde el planificador trate de ejecutarlos.

El Algoritmo 2 presenta el algoritmo de planificación considerando la degradación de prestaciones que sufre cada benchmark y está basado en la propuesta de planificación considerando el ancho de banda en toda la jerarquía de cache (Algoritmo 1). Para incluir el coeficiente de penalización se modifican tres pasos en el algoritmo. En primer lugar, el IABW se sustituye por el PIABW (IABW ponderado), que añade el coeficiente de penalización al BTR_{MM} de cada proceso para calcular las peticiones totales de memoria. En segundo lugar, la función de FITNESS también se modifica para que considere el coeficiente de penalización de cada proceso para su selección. Finalmente, cuando el proceso que maximiza la función fitness se elege para ejecutarse en el siguiente *quantum*, el ancho de banda restante se actualiza sustrayendo el BTR del proceso y su coeficiente de penalización.

En resumen, con las mejoras propuestas sobre el algoritmo de planificación, el PIABW calculado como el ancho de banda medio ideal para las cargas es mayor que el IABW que se calculaba previamente. Cuando los procesos seleccionados para su ejecución pertenecen al grupo de aplicaciones *poco sensibles*, el algoritmo trata de seleccionarlos de manera que el ancho de banda total requerido por ellos se acerque al PIABW. Se trata de un valor superior al IABW, con lo que su contención en el acceso a memoria debería ser mayor. Sin embargo, dado que los procesos seleccionados muestran poca degradación provocada por la contención en el acceso a memoria, la pérdida de prestaciones no debería ser significativa. Por el contrario, cuando los procesos seleccionados pertenecen al grupo de procesos *sensibles*, el coeficiente de penalización se aplica a la función fitness y en la actualización del ancho de banda restante, de manera que los procesos restantes seleccionados presentan un BTR menor que el PIABW y el IABW. Con ello, los procesos *sensibles* se ejecutan en situaciones con menos contención a priori, con lo que sus prestaciones deberían ser mayores. Es interesante observar que el algoritmo trata de aproximar el BTR_{MM} de cada *quantum* al PIABW calculado,

Algoritmo 2 Planificador considerando la degradación de prestaciones de los benchmarks

Requiere que los procesos que forman las cargas se envíen con su tiempo de ejecución y BTR_{MM}

$PIABW = \frac{\sum_{p=0}^{P}(BTR_{MM}^{p}+Coef.Penal.^{p})*T^{p}}{\frac{\sum_{p=0}^{P}T^{p}}{\#núcleos}}$

while haya procesos sin finalizar **do**
  Detiene los procesos en ejecución y los inserta al final de la cola de procesos
  for cada proceso P ejecutado durante el último *quantum* **do**
    for cada nivel L de la jerarquía con caches compartidas **do**
      Actualiza el BTR de P en el nivel de cache L
    end for
  end for
  $\text{BW}_{Restante}$ = PIABW, $\text{CPU}_{Restante}$ = núcleos
  Selecciona el primer proceso de la cola y actualiza $\text{BW}_{Restante}$ y $\text{CPU}_{Restante}$
  while $\text{CPU}_{Restante} > 0$ **do**
    Selecciona el proceso P que maximiza
    FITNESS(p) = $\frac{1}{\left|\frac{BW_{Restante}}{CPU_{Restante}}-(BW_{requerido}^{P}+Coef.Penal.^{p})\right|}$
    y actualiza $\text{BW}_{Restante}$ (sustrayendo $BW_{requerido}^{P}$ y $Coef.Penal.^{p}$) y $\text{CPU}_{Restante}$
  end while
  for cada nivel L de la jerarquía con caches compartidas **do**
    AVG_{BTR}(L)=$\frac{\sum BTR\ \ en\ \ L(i-1)}{\#Caches\ \ en\ \ L}$
    for cada cache en el nivel L **do**
      $\text{BW}_{Restante}$ = AVG_{BTR}(L), $\text{CPU}_{Restante}$ = núcleos que comparten la cache L
      while $\text{CPU}_{Restante} > 0$ **do**
        Selecciona el proceso P que maximiza la función FITNESS(p) y actualiza $\text{BW}_{Restante}$ y $\text{CPU}_{Restante}$
      end while
    end for
  end for
  Desbloquea los procesos seleccionados y los asocia al núcleo seleccionado.
  Duerme durante el *quantum*
end while

pero dado que los procesos añaden un coeficiente de penalización, es decir, declaran un BTR_{MM} mayor que el que realmente esperan usar, al ejecutar procesos *sensibles* el BTR real que se consume es menor que el PIABW. De este modo se espera reducir la contención de los procesos *sensibles* a expensas de un incremento ligero en los procesos *poco sensibles*, traduciendo esto en unas mejores prestaciones.

Capítulo 7

Evaluación del planificador propuesto

7.1. Diseño de las cargas y metodología de la evaluación

Para evaluar la efectividad de las propuestas se ha diseñado un conjunto de diez cargas con los benchmarks de la *suite* SPEC CPU2006. Las cargas 1-7 están formadas por ocho benchmarks, el doble que el número de núcleos, mientras que las cargas 8-10 se componen de doce benchmarks, triplicando el número de núcleos. La Tabla 7.1 muestra los benchmarks que componen cada carga y su IABW asociado. Como se puede observar, las cargas presentan valores de IABW entre 20 y 40 transacciones por microsegundo. Estos valores de IABW representan la zona de funcionamiento óptimo de los algoritmos de planificación que consideran el ancho de banda disponible para acceder a memoria. Si una carga se compone principalmente de benchmarks con bajos requisitos de acceso a memoria, el IABW será bajo, pero representará una carga donde la contención por acceso a memoria será baja y por tanto no tendrá sentido utilizar un algoritmo de planificación que considere el ancho de banda consumido. Por otra parte, las cargas con un IABW mayor que 40 estarán formadas mayormente por benchmarks con un BTR_{MM} alto, lo cual limita las capacidades del algoritmo de planificación que estará forzado a ejecutar concurrentemente aplicaciones con alta demanda de acceso a memoria principal.

Un aspecto importante de la metodología para evaluar los algoritmos de planificación es igualar el tiempo de ejecución de los benchmarks. Los tiempos varían ampliamente en función del benchmark desde unos veinte segundos, como *mcf*, hasta los cuatrocientos que requiere *tonto*. Esto es un inconveniente para evaluar la planificación, puesto que una política de planificación priorizando los trabajos más largos podría

Mixes	Benchmarks	IABW
Carga 1	*GemsFDTD, H264ref, Hmmer, Lbm, Lbm, Mcf, Tonto, Xalancbmk*	34.45
Carga 2	*Astar, Calculix, GemsFDTD, H264ref, Hmmer, Lbm, Mcf, Tonto*	23.46
Carga 3	*Astar, GemsFDTD, Hmmer, Lbm, Lbm, Mcf, Tonto, Xalancbmk*	37.13
Carga 4	*Astar, CactusADM, GemsFDTD, Lbm, Lbm, Mcf, Tonto, Xalancbmk*	39.37
Carga 5	*Astar, Bwaves, CactusADM, Lbm, GemsFDTD, Mcf, Tonto, Xalancbmk*	26.37
Carga 6	*Astar, DealII, GemsFDTD, H264ref, Lbm, Mcf, Namd, Sjeng*	24.31
Carga 7	*CactusADM, GemsFDTD, Mcf, Milc, Lbm, Leslie3d, Tonto, ZeusMP*	26.32
Carga 8	*Astar, Bzip2, DealII, Gcc, GemsFDTD, H264ref, Lbm, Lbm, Mcf, Mcf, Namd, Sjeng*	29.45
Carga 9	*Astar, Bwaves, CactusADM, CactusADM, DealII, Lbm, Lbm, Mcf, Soplex, Tonto, Xalancbmk, ZeusMP*	31.14
Carga 10	*Astar, Bwaves, CactusADM, DealII, GemsFDTD, Lbm, Lbm, Mcf, Milc, Sjeng, Tonto, Xalancbmk*	29.81

Tabla 7.1: Cargas

obtener las mejores prestaciones en la mayoría de cargas. Para solucionarlo, de cada benchmark únicamente se ejecuta el número de instrucciones necesario para alcanzar los dos minutos de ejecución cuando el benchmark se ejecuta en solitario. Dado que los benchmarks pueden presentar un tiempo de ejecución superior o inferior, el planificador será responsable cada *quantum* de actualizar las instrucciones ejecutadas por cada proceso y terminar aquel que alcance el número de instrucciones a ejecutar, o relanzar una nueva instancia del benchmark si éste ha finalizado su ejecución sin completar el número de instrucciones determinado.

Se ha evaluado el rendimiento de los siguientes algoritmos de planificación:

- Algoritmo de planificación basado en el algoritmo de planificación usado por Linux.
- Algoritmo de planificación PcM, que considera la contención en el acceso a memoria principal [XWY10].
- Algoritmo de planificación PcJ, que considera la contención en el acceso a la jerarquía de cache.

- Algoritmo de planificación PcD, que consider la contención en el acceso a la jerarquía de cache y la degradación de cada benchmark.

Los algoritmos se han implementado como planificadores a nivel de usuario. Estos programas se ejecutan por encima del planificador del sistema operativo, pero se encargan de bloquear o desbloquear y asociar los procesos a los núcleos de manera que se fuerza al planificador del sistema operativo a realizar la planificación deseada. Los planificadores implementados comparten la mayor parte del código, lo cual favorece una comparación más justa y simplemente difieren en la función de selección y ubicación de los procesos a ejecutar. En el algoritmo basado en el planificador de Linux, en cada *quantum* se seleccionan todos los procesos para su ejecución y el planificador del sistema operativo decide cuales ejecutar en cada momento. El algoritmo de planificación PcM sólo selecciona los procesos y deja la ubicación en núcleos concretos al sistema operativo. Por su parte, las dos propuestas PcJ y PcD presentadas en este trabajo siguen los algoritmos descritos en el Capítulo 6, seleccionando y asignando los procesos a los núcleos más apropiados.

7.2. Prestaciones de las propuestas

La Figura 7.1 muestra la aceleración obtenida por los planificadores PcM, PcJ y PcD utilizando como base el tiempo de ejecución con la planificación de Linux. Se puede observar que el planificador PcM mejora las prestaciones del planificador en todas las cargas diseñadas. La aceleración alcanzada varía entre el 1.64 % y el 5.22 %, con un valor medio del 3.25 %. La primera propuesta de planificación presentada, que considera el ancho de banda a lo largo de toda la jerarquía de memoria (PcJ) mejora las prestaciones de la planificación PcM en todas las cargas. La aceleración obtenida varía entre el 3.18 % y el 7.26 %, con una media del 4.86 %. A la vista de los resultados obtenidos, resulta interesante considerar el ancho de banda a lo largo de toda la jerarquía de memoria para realizar la planificación, pues la aceleración respecto a la planificación PcM se mejora un 49 %. A su vez, la planificación considerando la degradación del IPC que sufre cada benchmark (PcC), que es una propuesta incremental sobre el PcJ, mejora las prestaciones de las propuestas anteriores. Ofrece una aceleración que varía entre el 3.66 % y el 9.56 %, con un valor medio del 6.55 %. Este valor significa una mejora del 34 % respecto a la planificación PcJ. Es más, en la mitad de las cargas la planificación PcD consigue triplicar las prestaciones de PcM.

El motivo principal que explica el incremento de prestaciones de la planificación PcJ en comparación con la planificación PcM es que disminuye la contención en el acceso a cada nivel de la cache equilibrando las transacciones totales entre todas las caches disponibles en cada nivel. En la plataforma experimental, donde la jerarquía es

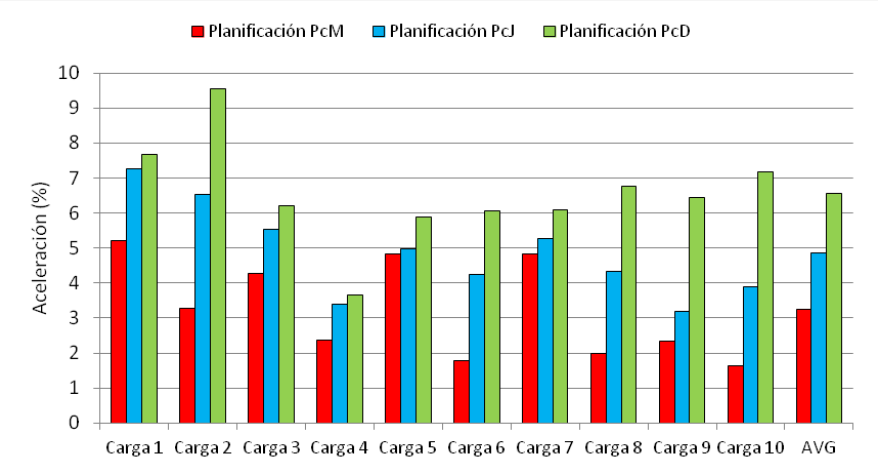


Figura 7.1: Aceleración respecto a la planificación de Linux

de dos niveles y el último nivel está formado por dos caches, la mejora de prestaciones se basa ubicar los procesos en los núcleos de manera que se equilibren las peticiones que llegan a cada una de las dos caches.

Para estimar si se consigue distribuir equitativamente las transacciones entre las dos caches, se compara el BTR que llega a cada una de ellas, provocado por fallos en las caches del nivel uno. La Figura 7.2 presenta los resultados obtenidos para las planificaciones PcM y PcJ en forma de histograma. El histograma representa la frecuencia de la diferencia del BTR_{L2} entre las dos caches L2 en cada una de las cargas analizadas. Los resultados se presentan en intervalos de 25 transacciones por microsegundo. Cuanto más alta sea la frecuencia de los intervalos más bajos, mejor será el equilibrio del BTR entre las dos caches. Se espera que un mejor equilibrio del BTR implique una menor contención en el acceso a la cache L2 y por tanto el planificador debería alcanzar

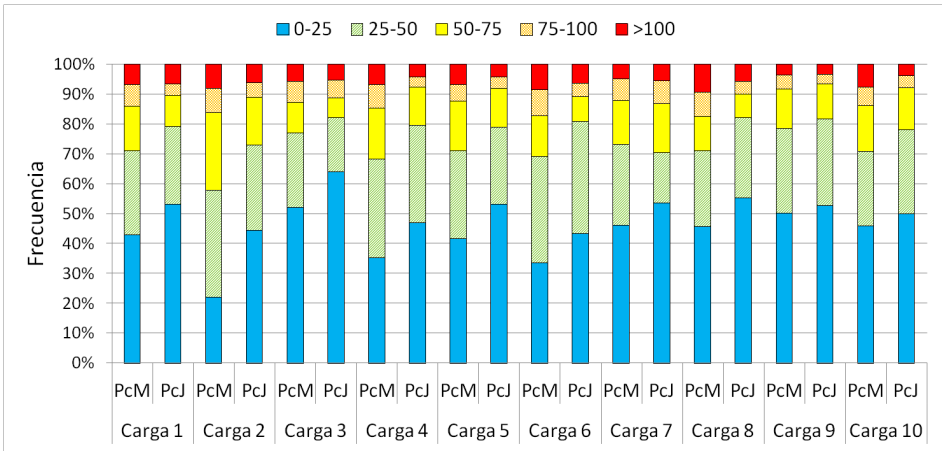


Figura 7.2: Diferencia del BTR_{L2} entre las dos caches

mejores prestaciones.

Por ejemplo, si observamos la carga 1 se comprueba que para el planificador PcM, el 42 % de las veces la diferencia entre el BTR_{L2} que recibe cada una de las dos caches es menor que 25 transacciones por microsegundo (barra inferior). La barra inmediatamente superior indica que alrededor del 30 % de las veces la diferencia se encuentra entre 25 y 50 transacciones por microsegundo, y así sucesivamente. Por su parte, en la planificación PcJ un 53 % de las veces la diferencia se encuentra entre 0 y 25 transacciones por microsegundo. Esto implica que la contención sea menor en el acceso a la L2 y permite obtener mejores prestaciones globales.

Como se esperaba, los resultados muestran una cierta correlación entre la aceleración de la carga y la distribución equilibrada de los accesos a L2. De hecho, la carga 2, que con la planificación PcJ presenta una de las mayores aceleraciones respecto a la planificación PcM es la que ofrece una mejora más significativa en el equilibrio de los accesos a L2. Esto se puede apreciar claramente en el intervalo de 0 a 25 transacciones por microsegundo, donde la frecuencia en este intervalo pasa de un 21 % a un 43 %. Por el contrario, la carga 7 que presenta una de las menores aceleraciones con respecto a la planificación PcM, ofrece también las menores diferencias en el equilibrado de los accesos a las caches L2. Sin embargo, en algunas cargas como la 9, las frecuencias que presenta el histograma son muy similares entre las planificaciones PcM y PcJ. Esto se puede explicar por los resultados de la Figura 7.3, donde se muestra la media y la varianza de la diferencia del BTR entre las dos caches. Si bien el histograma muestra frecuencias similares, la media muestra como el equilibrado con la planificación PcJ es mejor, y por ello se mejoran las prestaciones de la carga.

Con el objetivo de proporcionar una mejor comprensión del equilibrado del BTR vamos a analizar la diferencia de los accesos entre las dos caches de manera dinámica, durante la ejecución de la carga 2 donde la planificación PcJ dobla las prestaciones

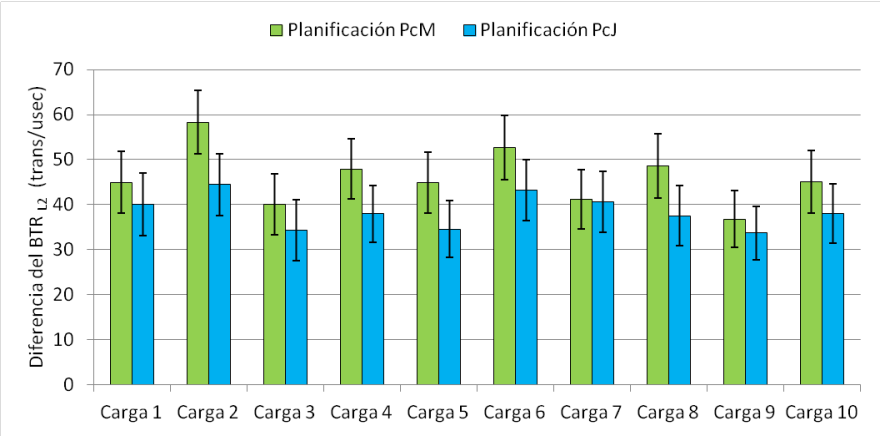


Figura 7.3: Media y varianza de la diferencia del BTR_{L2} entre las dos caches

respecto a la planificación PcM. La Figura 7.4 muestra la diferencia en el BTR_{L2} entre las dos caches de manera dinámica para los primeros 160 *quanta* de ejecución. La figura permite observar como el equilibrio del BTR con la planificación PcJ es mejor, pues los picos de diferencia del BTR son menores y considerablemente más pequeños.

La Figura 7.5 muestra la diferencia del BTR_{L2} durante los primeros 1375 *quanta* de ejecución en los planificadores PcM, PcJ y PcD. Una observación interesante es que los picos de diferencia del BTR (con fondo gris en la figura), que en esta carga son principalmente provocados por el benchmark *mcf*, se producen antes con la planificación PcJ, sin que este adelantamiento en el tiempo provoque incrementos en los picos, que de hecho también son ligeramente reducidos. Nótese que con la planificación PcM la línea que marca la diferencia de 50 transacciones por microsegundo queda oculta por la variación del BTR, mientras que con la planificación PcJ son varios los intervalos donde el equilibrado consigue situarse por debajo de las 50 transacciones por microsegundo durante un centenar de ciclos consecutivos. La figura también muestra como la planificación PcD consigue adelantar aún más los picos provocados por el benchmark *mcf*, lo que en definitiva significa acelerar la ejecución, sin que esto provoque un aumento en las diferencias del BTR_{L2} entre las caches, ya que con respecto a PcJ las diferencias del BTR_{L2} menores que 50 transacciones por microsegundo también se ven reducidas.

Finalmente, la Figura 7.6 muestra la aceleración media de los benchmarks comparando la planificación PcJ frente a la planificación PcD para cada carga. En concreto, se muestra el porcentaje de los benchmarks que forman la carga que obtiene una aceleración en los intervalos indicados. De este modo, la barra inferior muestra la frecuencia con la que utilizando la planificación PcD los benchmarks obtienen una aceleración negativa del IPC (deceleración) entre el -5 % y el -2.5 % respecto a la planificación PcJ. La siguiente barra muestra el porcentaje que ofrecen una aceleración entre el -2.5 % y

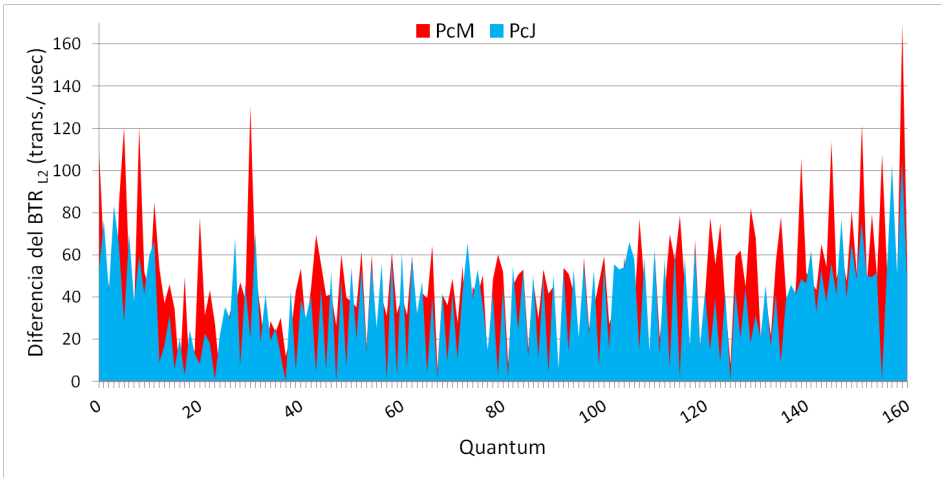


Figura 7.4: Evolución de la diferencia del BTR_{L2} los primeros 160 *quanta* de ejecución

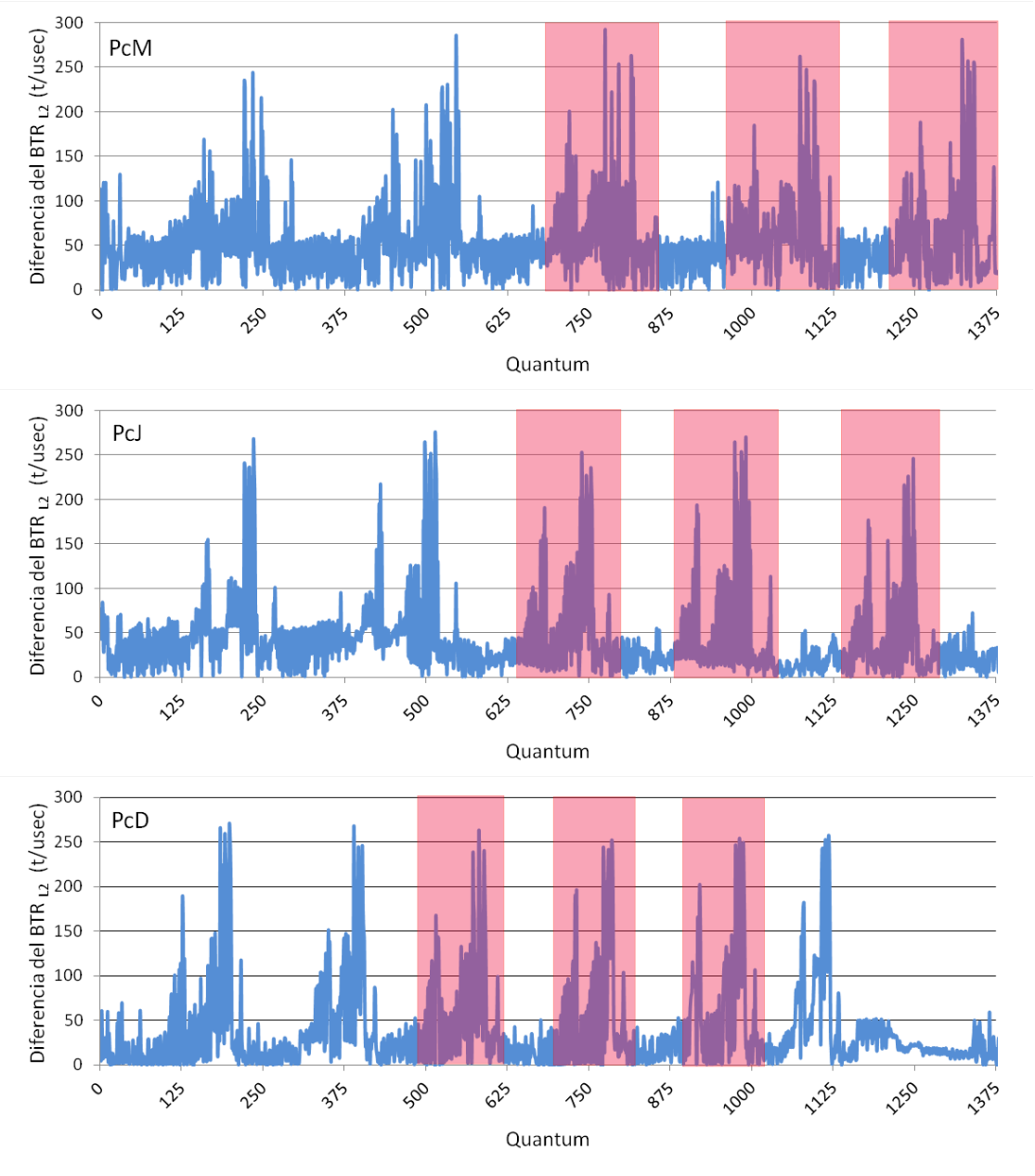


Figura 7.5: Comparación de la diferencia del BTR_{L2} con los diferentes algoritmos de planificación

el 0 %. A partir de aquí, las barras muestran aceleraciones positivas en los intervalos de 0 % al 2.5 %, del 2.5 % al 5 % y superior al 5 %.

Se puede observar, que en seis de las cargas que corresponden con las cargas que ofrecen mejor aceleración respecto a la planificación PcJ, el porcentaje de benchmarks con aceleración positiva supera el 60 % y cuatro cargas disponen de benchmarks que consiguen una aceleración superior al 5 %. En cargas donde la aceleración respecto a la planificación PcJ es menor, el número de benchmarks con aceleración positiva y

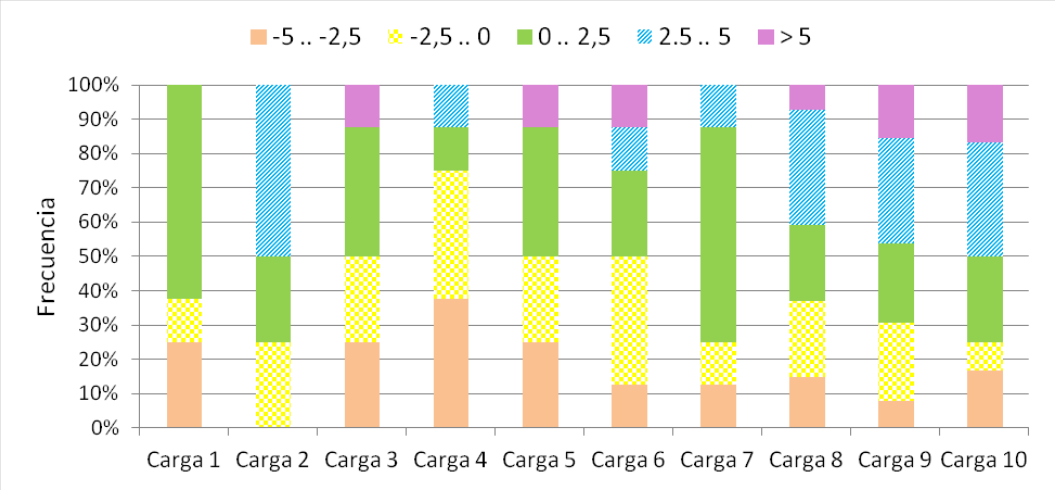


Figura 7.6: Porcentaje de las aplicaciones que utilizando el planificador PcD presentan una aceleración/deceleración del IPC en cada intervalo respecto al planificador PcJ

negativa se equilibra, sin embargo, siguen apareciendo benchmarks con aceleraciones superiores al 5 % (por ejemplo, en la carga 6), lo que finalmente permite una aceleración global en el tiempo de ejecución de la carga completa. Un caso singular es el que presenta la carga 4, donde el porcentaje de aplicaciones con aceleración negativa es muy superior al que presentan aceleraciones positivas y, sin embargo, se sigue produciendo una mejora en el tiempo de ejecución de la carga. Esto se debe a que el tiempo de finalización de los benchmarks individuales difiere más con la planificación PcJ que con la PcD, lo que provoca que algunos benchmarks puedan ejecutarse más tiempo con un mayor número de núcleos desocupados. Esto hace que la contención en ese momento sea inferior y por tanto, el IPC del benchmarks pueda ser mejor, pero se traduce en una menor utilización de los núcleos y provoca que el tiempo de ejecución de la carga sea mayor que con la planificación PcD, donde la finalización de todos los benchmarks está mucho más agrupada.

Capítulo 8

Conclusiones, Publicaciones y Trabajo Futuro

8.1. Conclusiones

El presente trabajo se ha centrado en la planificación de procesos para procesadores multinúcleo, a partir de la optimización del ancho de banda consumido a lo largo de la jerarquía de memoria del procesador.

En primer lugar, el trabajo ha presentado una serie de experimentos que muestran la importancia de la LLC en las prestaciones de las cargas actuales. Para ello, se ha diseñado un conjunto de algoritmos destinados a a obtener las características de las LLC, tanto la geometría (número de conjuntos, número de vías y tamaño de línea) como la arquitectura (número de caches que forman cada nivel y número de núcleos que comparten cada cache). Estos algoritmos hacen uso de aspectos avanzados de la arquitectura como son los contadores de prestaciones y las *huge pages*, de los cuales se han descrito tanto los aspectos teóricos como prácticos.

Basados en los algoritmos anteriores, se ha diseñado una serie de experimentos para analizar el comportamiento de la degradación de prestaciones de los benchmarks por contención en el acceso a memoria principal y a la LLC en diferentes situaciones de carga. El análisis de los resultados ha demostrado que la contención puede provocar pérdida de prestaciones tanto en el acceso a memoria principal como en el acceso a la LLC. Además, se espera que la contención en la LLC se incremente en futuros procesadores, pues la actual tendencia de la industria es a incrementar el número de núcleos computacionales e hilos de ejecución que estos implementan.

La degradación de prestaciones observada por contención en el acceso a la LLC, muestra la necesidad de una política de planificación que considere la contención a lo largo de toda la jerarquía de memoria y no únicamente en el acceso a la memoria principal. Para resolverlo, se han diseñado dos políticas de planificación que consideran

todos los puntos de contención de la jerarquía, y siguiendo una estrategia top-down (desde la memoria principal a la cache L1) seleccionan y ubican los procesos en los núcleos, de manera que se distribuyen equitativamente todos los accesos en cada nivel de cache entre las diferentes caches disponibles. La evaluación de prestaciones realizada sobre el primer planificador propuesto muestra que las prestaciones se incrementan un 49 % al compararlo con un planificador que únicamente considera la contención en el acceso a memoria principal. Si se compara con el tiempo de ejecución de las cargas sobre Linux, la planificación mejora un 4.86 %.

El segundo planificador propuesto tiene en cuenta la degradación que sufren los benchmarks en situaciones con un consumo de ancho de banda alto, como las creadas por los planificadores que maximizan la utilización de la jerarquía de memoria. Este planificador clasifica los benchmarks en dos tipos: *sensibles* a la contención en el acceso a memoria principal e *poco sensibles* a la contención en el acceso a memoria principal. Los benchmarks *sensibles* se planifican para ser ejecutados en situaciones donde el ancho de banda total consumido sea menor. Como contrapartida, los benchmarks *poco sensibles* se ejecutan en situaciones de mayor consumo de ancho de banda. No obstante, la mejora obtenida por la ejecución de los benchmarks *sensibles* compensa las ligeras pérdidas de prestaciones de los benchmarks *poco sensibles*, con lo que el resultado final es una mejora de prestaciones. La evaluación de prestaciones demuestra que la propuesta mejora un 6.55 % el tiempo de planificación de Linux, y comparando la aceleración con la que se obtenía con la planificación PcJ, la mejora es del 35 %.

8.2. Publicaciones

El trabajo de investigación que se expone ha dado lugar a una publicación en un congreso internacional:

- J. Feliu, J. Sahuquillo, S. Petit, y J. Duato, Understanding Cache Hierarchy Contention in CMPs to Improve Job Scheduling, *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 508-519, ISBN: 987-0-7695-4675-9, Shanghai, China, May 2012. (Indicios de calidad Core A).

Y se ha aceptado otra publicación en un congreso nacional:

- J. Feliu, J. Sahuquillo, S.r Petit, y J. Duato, Planificación Considerando el Ancho de Banda de la Jerarquía de Cache, *XXIII Jornadas de Paralelismo SARTECO*, Elche, Septiembre 19-21, 2012. (Aceptado y pendiente de publicación).

Además, dos publicaciones más han sido ya enviadas y se encuentran en proceso de revisión para su publicación, si procede, en una revista y un congreso internacionales:

- J. Feliu, J. Sahuquillo, S. Petit, y J. Duato, Cache-Hierarchy Aware Scheduling in CMPs, *IEEE Transactions on Parallel and Distributed Systems*, 2012. Indicios de calidad: Journal situado en el primer tercil del Journal of Citation Report (JCR). Enviado y pendiente de notificación.

- J. Feliu, J. Sahuquillo, S. Petit, y J. Duato, Using huge pages and performance counters to determine the LLC architecture, *24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2012. Enviado y pendiente de notificación. (Indicios de calidad Core B).

8.3. Trabajo Futuro

Dado que la tendencia actual de la industria se basa en aumentar el número de núcleos y ampliar la jerarquía de cache, se prevé que el problema de la contención se incremente en futuros procesadores. Por tanto, muchos esfuerzos investigadores se concentran en algoritmos de planificación que limiten la contención en el acceso a la jerarquía de memoria. Por otro lado, el aumento de núcleos va a necesitar del diseño de planificadores novedosos que garanticen un cierto nivel de rendimiento limitando el consumo global.

La investigación no debe limitarse al estudio de los sistemas actuales, sino que debe prever las características relevantes de los procesadores futuros. Dos tendencias parecen claras al observar los prototipos de la industria. Por un lado, muchos procesadores futuros serán *manycore*, es decir, incluirán desde varias decenas a centenares de núcleos. Por otro lado, incorporarán núcleos heterogéneos, lo que significa que cada núcleo presentará prestaciones y consumos energéticos diferentes.

Para poder estudiar estos sistemas actualmente no disponibles, se utilizará el simulador Multi2Sim, diseñado en el Grupo de Arquitecturas Paralelas (GAP) de la Universidad Politécnica de Valencia, que cuenta con el aval de numerosas publicaciones de grupos internacionales y dispone del apoyo de grandes corporaciones como Samsung y AMD. Mediante la simulación nos será posible modelar las características de estos sistemas *manycore* heterogéneos para diseñar algoritmos de planificación eficientes, es decir, orientados tanto a rendimiento como a consumo.

Dado el dinamismo y las nuevas capacidades de los sistemas del futuro también nos proponemos explotar la cooperación *hardware/software* para proponer planificadores más eficientes y dinámicos. Las propuestas se basarán en dotar al procesador de soporte *hardware* específico que pueda monitorizar las tareas en ejecución. Los resultados de esta monitorización ayudarán al planificador a ubicar los procesos, de manera dinámica, en los núcleos que mejor se adapten a la carga de trabajo de cada proceso, incrementando las prestaciones mientras se reduce el consumo.

Bibliografía

[ANP03] C.D. Antonopoulos, D.S. Nikolopoulos, and T.S. Papatheodorou. Scheduling algorithms with bus bandwidth considerations for smps. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 547 –554, oct. 2003.

[ANP04] Christos D. Antonopoulos, Dimitrios S. Nikolopoulos, and Theodore S. Papatheodorou. Realistic workload scheduling policies for taming the memory bandwidth bottleneck of smps. In *In Proc. of the 2004 IEEE/ACM International Conference on High Performance Computing (HiPC'2004)*, pages 286–296, 2004.

[BZFK10] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. A case for numa-aware contention management on multicore systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 557–558, New York, NY, USA, 2010. ACM.

[ENBSH11] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Cache pirating: Measuring the curse of the shared cache. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 165 –175, sept. 2011.

[FBZ10] Alexandra Fedorova, Sergey Blagodurov, and Sergey Zhuravlev. Managing contention for shared resources on multicore processors. *Commun. ACM*, 53:49–57, February 2010.

[GDTF+10] J. Gonzalez-Dominguez, G.L. Taboada, B.B. Fraguela, M.J. Martin, and J. Tourio. Servet: A benchmark suite for autotuning on multicore clusters. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –9, april 2010.

[HMR+00] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir. Introducing the ia-64 architecture. *Micro, IEEE*, 20(5):12 –23, sep/oct 2000.

[Int] Intel. Intel Xeon Processor X3320. http://ark.intel.com/products/33931/Intel-Xeon-Processor-X3320-%286M-Cache-2_50-GHz-1333-MHz-FSB%29.

[JJN08] S Jarp, R Jurga, and A Nowak. Perfmon2: a leap forward in performance monitoring. *Journal of Physics: Conference Series*, 119(4):042017, 2008.

[KCS04] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 111–122, Washington, DC, USA, 2004. IEEE Computer Society.

[KK06] Evangelos Koukis and Nectarios Koziris. Memory and network bandwidth aware scheduling of multiprogrammed workloads on clusters of smps. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems - Volume 1*, ICPADS '06, pages 345–354, Washington, DC, USA, 2006. IEEE Computer Society.

[KSB+09] R. Kuppuswamy, S.R. Sawant, S. Balasubramanian, P. Kaushik, N. Natarajan, and J.D. Gilbert. Over one million tpcc with a 45nm 6-core xeon cpu. In *Solid-State Circuits Conference - Digest of Technical Papers, 2009. ISSCC 2009. IEEE International*, pages 70 –71,71a, feb. 2009.

[KSCJ10] D. Kaseridis, J. Stuecheli, Jian Chen, and L.K. John. A bandwidth-aware memory-subsystem resource management using non-invasive resource profilers for large cmp systems. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1 –11, jan. 2010.

[Man04] Stefan Manegold. The calibrator, a cache-memory and tlb calibration tool. webpage, June 2004. available at `http://homepages.cwi.nl/~manegold/Calibrator/`.

[MS] Larry McVoy and Carl Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pages 23–23.

[QLMP06] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A Case for MLP-Aware Cache Replacement. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 167–178, Washington, DC, USA, 2006. IEEE Computer Society.

[SB92] Rafael H. Saavedra-Barrera. Cpu performance evaluation and execution time prediction using narrow. Technical report, Berkeley, CA, USA, 1992.

[SDR02] G.E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pages 117 – 128, feb. 2002.

[SKE+09] M. Sato, I. Kotera, R. Egawa, H. Takizawa, and H. Kobayashi. A cache-aware thread scheduling policy for multi-core processors. In *Parallel and Distributed Computing and Networks (PDCN), 2009 IASTED 8th International Conference on*, pages 109 –114, feb. 2009.

[SKT+05] B. Sinharoy, R N. Kalla, J M. Tendler, R J. Eickemeyer, and J B. Joyner. POWER5 System Microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505–521, 2005.

[TMV+11] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary-Lou Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *ISCA*, 2011.

[WW09] P. Weisberg and Y. Wiseman. Using 4kb page size for virtual memory is obsolete. In *IEEE International Conference on Information Reuse Integration*, pages 262 –265, 2009.

[XWY10] Di Xu, Chenggang Wu, and Pen-Chung Yew. On mitigating memory bandwidth contention through bandwidth-aware scheduling. In *PACT*, pages 237–248, 2010.

[ZBF10] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 129–142, New York, NY, USA, 2010. ACM.

[ZJS10] Eddy Z. Zhang, Yunlian Jiang, and Xipeng Shen. Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? *SIGPLAN Not.*, 45(5):203–212, January 2010.