Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Evaluación de la jerarquía de cache en procesadores multinúcleo

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Julio Antonio Vivas Vivas

Tutor: Salvador Vicente Petit Martí

Cotutor: Julio Sahuquillo Borrás

Curso: 2013/2014

Resumen

Este proyecto se centra en la evaluación de diferentes jerarquías de cache reales (basadas en los últimos diseños de Intel) en procesadores multinúcleo. Los estudios se realizan mediante el simulador Multi2Sim para la obtención de información sobre el comportamiento de cada nivel de la jerarquía cuando el procesador ejecuta cargas multiprogramadas, que constan de diversas aplicaciones ejecutándose en núcleos distintos. La información obtenida es de diversa índole y abarca todos los niveles de la jerarquía. Posteriormente se realiza un análisis de los datos obtenidos.

Inicialmente se estudia las prestaciones de las aplicaciones individuales, ejecutándolas de manera aislada en un procesador con un solo núcleo; para ir añadiendo más núcleos y complejidad a la jerarquía. El objetivo final es simular el comportamiento de jerarquías similares a las que se pueden encontrar en los procesadores comerciales recientes. Los resultados de los distintos estudios nos permite analiza cómo afecta a las prestaciones de cada aplicación el competir en el acceso a las caches compartidas con otras aplicaciones.

Palabras clave: jerarquía de cache, procesador multinúcleo, rendimiento.

Abstract

This project focuses on the evaluation of different real cache hierarchies (based on the latest Intel designs). It uses the Multi2Sim simulator to obtain information about the utilization of each level of the hierarchy when the processor executes several applications. The information obtained covers all levels of the cache hierarchy.

Initially the hierarchy of a single core processor will be studied. Then, we add more cores and increase the complexity of the cache hierarchy, in order to add more cores and complexity to the hierarchy. The ultimate goal is to simulate hierarchies close to those of recent commercial processors. Comparing the results of the different studies, we can infer how cache hierarchy contention impacts on the performance of each application.

Keywords: cache hierarchy, multicore processor, performance.

Agradecimientos

Este trabajo no habría sido posible sin la ayuda y orientación constantes de Salvador Petit y Julio Sahuquillo. Desde estas líneas quiero expresarles mi gratitud por la paciencia mostrada a lo largo de estos últimos meses, desde las dudas iniciales sobre el alcance del estudio hasta las revisiones finales del texto, así como por los consejos para la redacción y presentación de los resultados experimentales.

Aunque oficialmente sólo puedan figurar como tales los dos profesores titulares de la universidad, este proyecto ha tenido el honor de contar con un tercer tutor oficioso. Quiero agradecer especialmente a Alejandro Valero por su incansable ayuda, consejo y apoyo a la hora de comprender las posibilidades de Multi2Sim.

Tabla de contenidos

1. Introducción y objetivos

En la actualidad, el avance en la mejora del rendimiento de los computadores se encuentra en una fase crítica. La llamada Ley de Moore, enunciada a mediados de la década de los 60 del siglo pasado por el cofundador de Intel Gordon E. Moore, y según la cual la capacidad de los sistemas informáticos se duplica cada año, tuvo que ser reformulada por su propio autor, ya en una fecha tan temprana como 1975, para aumentar el intervalo hasta los 18 meses, a la vista de la ralentización en la velocidad de integración de los circuitos.

Hoy en día, en pleno siglo XXI, la creciente dificultad en el aumento de la escala de integración ha obligado a la industria de los computadores a encontrar vías alternativas que permitan mantener en vigor las afirmaciones de Moore y, con ello, responder a la creciente demanda tecnológica de nuestra sociedad.

Uno de los campos donde probablemente el margen de mejora sea mayor reside en la optimización del sistema de memoria de los computadores.

Este estudio, enmarcado en el ámbito del Trabajo Fin de Titulación del Grado en Ingeniería Informática de la Universidad Politécnica de Valencia, tiene como objeto analizar algunos de los aspectos clave –con las limitaciones inherentes a un proyecto de este tipo– en el comportamiento de un componente crucial del sistema de memoria: la jerarquía de cache.

1.1 Objetivos

El objetivo principal es evaluar el rendimiento de la jerarquía de cache de los computadores actuales y obtener así conclusiones que permitan identificar problemáticas y soluciones asociados a la misma, a través de indicadores cuantitativos como distribución de aciertos y fallos e índices de prestaciones como el IPC (*Instructions Per Cycle*) y el MPKI (*Misses Per Kilo-Instruction*).

Este objetivo principal se desglosa en objetivos específicos, entre los que cabe destacar:

- Modelar sistemas multinúcleo para profundizar en el conocimiento de estas arquitecturas.

- Comprobar el impacto que supone la optimización de la jerarquía de cache sobre el rendimiento general de los sistemas multinúcleo.

1.2 Metodología

El proyecto ha empleado una metodología de trabajo basada en el uso del software Multi2Sim, un simulador de procesadores multinúcleo muy popular en los ámbitos industrial y académico. El simulador nos permite modelar diferentes jerarquías de memoria cache así como núcleos de procesadores actuales de Intel. Una vez modelado es posible realizar sucesivos experimentos de rendimiento para las diferentes jerarquías, obtener resultados y realizar un análisis estadístico de los datos obtenidos en las distintas simulaciones.

El proyecto se ha dividido en varias fases, correspondientes a los distintos experimentos realizados. Cada una de ellas se subdivide a su vez en cinco etapas diferenciadas:

- Modelado del sistema a simular. Cada computador y arquitectura a simular ha de ser desarrollada previamente mediante los parámetros que proporciona Multi2Sim.

- Modificación del código fuente del simulador. A menudo ha sido necesario desarrollar código adicional para el simulador, cuando se precisaba modelar características no disponibles en Multi2Sim. Dada la gran complejidad del simulador, estas modificaciones han supuesto una dificultad añadida al trabajo realizado.

- Lanzamiento de simulaciones. Los sistemas modelados, junto con las cargas con las que trabajarán, se simulan en un clúster de computadores real. Cada simulación supone entre pocas horas y varios días de ejecución en el clúster, según su ocupación.

- Toma de resultados. Conexión remota al clúster para extraer los datos devueltos por la simulación, separando la información importante de la superflua que carezca de validez para el estudio.

- Tratamiento de los datos. Análisis estadístico de los datos obtenidos, generación de gráficas comparativas y extrapolación de conclusiones en base a los mismos.

Para la realización del estudio ha sido necesario afrontar distintos problemas que le han ido sumando dificultad: i) la complejidad del propio simulador, cuya fase de aprendizaje (aunque solo sea una parte de éste) requiere consultar gran cantidad de información y de código, y se estima en más de 6 meses, ii) la gran cantidad de información suministrada por Multi2Sim, que obliga a un análisis y selección cuidadosa de los datos obtenidos, iii) el hecho de realizar los experimentos en un clúster real, compartido por múltiples usuarios y sometido a las vicisitudes habituales en todo sistema informático de gran magnitud, lo que extiende la duración de las simulaciones, etc. A pesar de ello, se han conseguido los objetivos planteados inicialmente del proyecto, cuyos resultados se presentan en los capítulos finales del presente texto.

1.3 Resumen

En este capítulo se ha detallado la motivación del proyecto y presentado los objetivos a alcanzar. Tras realizar una breve reseña de la situación histórica y actual de la industria de los computadores, se han descrito las principales metas propuestas, para finalizar explicando la metodología de trabajo a seguir para la consecución de las mismas.

El resto de este trabajo se organiza como sigue. En el capítulo 2, se presenta la base teórica y los conceptos necesarios para la comprensión y seguimiento del trabajo realizado. En el capítulo 3, se presenta el entorno experimental en que se han llevado a cabo las diferentes pruebas del estudio. El capítulo 4 detalla los sucesivos experimentos realizados. Por último, el capítulo 5 presenta las conclusiones principales a partir de los resultados obtenidos.

2. La jerarquía de cache en el chip

A pesar de que un lector iniciado en el ámbito de la ingeniería de computadores se encontrará familiarizado con el concepto de memoria cache, es conveniente comenzar por describir algunos aspectos meramente teóricos relativos a la arquitectura de memoria de los sistemas informáticos, así como aclarar algunos de los términos que se utilizan a lo largo del texto. Para una lectura general sobre estructura de computadores, se puede consultar bibliografía específica como el libro de David A. Patterson y John L. Hennessy [1], así como el volumen de los mismos autores sobre arquitectura de computadores [2].

En un computador ideal, el procesador contaría con una cantidad ilimitada de memoria rápida. Por desgracia, no existe tecnología alguna capaz de cumplir estos requisitos. Se impone por tanto una organización de la memoria en distintos niveles, donde la memoria más rápida se encuentra lo más cerca posible del procesador, para atender sus peticiones de forma óptima.

El éxito de la jerarquía de memoria en los computadores se debe a la validez del *principio de localidad*. Este principio tiene dos vertientes. Por un lado, observamos una localidad *temporal*, ya que los elementos del software (variables, objetos, etc.) accedidos son generalmente invocados de nuevo en un intervalo de tiempo corto. Por otro lado, se da también una localidad *espacial*, ya que los elementos con direcciones de memoria cercanas también tienden a referenciarse en intervalos cortos de tiempo (esta segunda faceta del principio de localidad se ve potenciada con la organización en bloques de la memoria).

La jerarquía de memoria demuestra ser una solución eficiente al combinar tecnologías de características dispares. Las técnicas más rápidas y eficientes, y por tanto más caras, se sitúan en las cercanías del procesador, mientras que las tecnologías menos eficientes (y por consiguiente más económicas) se usan para tareas de almacenamiento masivo en los niveles más alejados del núcleo.

2.1 Memorias cache

Las memorias cache conforman los primeros niveles de la jerarquía de memoria, inmediatamente después de los registros propios del procesador. En informática, el término *cache* se utiliza a menudo para referirse a información almacenada para ser reutilizada ("cache de nombres", "cache de disco", etc.). Por el principio de localidad y la propia estructura de la jerarquía de memoria, los primeros niveles han de contener la información que con más probabilidad vaya a requerir el procesador a corto plazo. Esta probabilidad decrece a medida que se desciende por los diferentes niveles de la jerarquía.

Las memorias cache, fabricadas generalmente con tecnología SRAM, cuentan con una latencia de acceso de entre 0.1 y 10ns. Es decir, más de cien veces inferior a la de la memoria principal. Su capacidad varía desde varios *kilobytes* para el nivel 1, actualmente integrado dentro del propio núcleo del procesador, hasta decenas de *megabytes* para las memorias cache de nivel 3. En contrapartida, los módulos de memoria principal actualmente existentes en el mercado proporcionan *gigabytes* de capacidad, lo cual es inviable cuando se trata de memorias cache SRAM.

Las memorias cache se organizan internamente en bloques. Cuando se accede a una memoria cache, se considera acierto (*hit*) si los datos solicitados están contenidos en un bloque de la misma, y fallo (*miss)* en caso contrario. Si se produce un fallo, se solicita a los niveles inferiores de la jerarquía una copia del bloque que contiene la información solicitada. Esta copia normalmente reemplazará uno de los bloques de la cache donde ocurrió el fallo.

Una *función de correspondencia* o *mapping* asigna los bloques a almacenar en una memoria cache en función de su dirección a una o más líneas de cache. Cada línea está compuesta por el suficiente número de celdas SRAM para contener un bloque completo. Las memorias cache se pueden clasificar en tres tipos distintos atendiendo a su función de correspondencia:

- Directas. Cada bloque que se ubica en la memoria cache tiene asignada una única línea, no pudiendo ubicarse en ninguna otra línea aunque estas estuvieran vacías.

- Completamente asociativas. Cada bloque de memoria principal puede ubicarse en cualquier línea de la cache.

- Asociativas por conjuntos de N vías. La cache se encuentra dividida en conjuntos (*sets*) formados por N líneas cada uno. Cada bloque de memoria principal está asignado a un conjunto determinado, pudiendo ocupar cualquier de las vías que lo forman.

Las *políticas de reemplazo* dictaminan qué bloque de la cache debe ser sustituido por el bloque ubicado por causa del fallo. Existen diferentes políticas de reemplazo, siendo la política LRU (*Least Recently Used*) una de las más empleadas. El algoritmo LRU se fundamenta en reemplazar el bloque que lleve más tiempo sin referenciarse, para lo cual se asignan contadores a cada uno de los bloques.

Por último, recordar que las memorias cache pueden ser *unificadas*, cuando contienen a la vez datos e instrucciones, o *duales*, cuando existen caches separadas para cada tipo de información. Esta última configuración se denomina también arquitectura *Harvard*.

2.2 Jerarquía de cache en chip en los procesadores actuales

La figura 2.2 muestra un esquema de la jerarquía de cache en un procesador actual, concretamente el modelo Intel® Core™ i7-4702EC, lanzado al mercado en el primer cuatrimestre de 2014. Se ha tratado de reflejar la diferencia de capacidad existente entre los módulos de memoria que componen los distintos niveles de la jerarquía de cache. Nótese que los distintos niveles de cache se denominan (como abreviación) con una letra L (del inglés *level*) seguida del número de nivel: L1 es el nivel de cache más cercano al procesador, L2 es el siguiente, etc. A lo largo del texto se ha utilizado esta terminología.

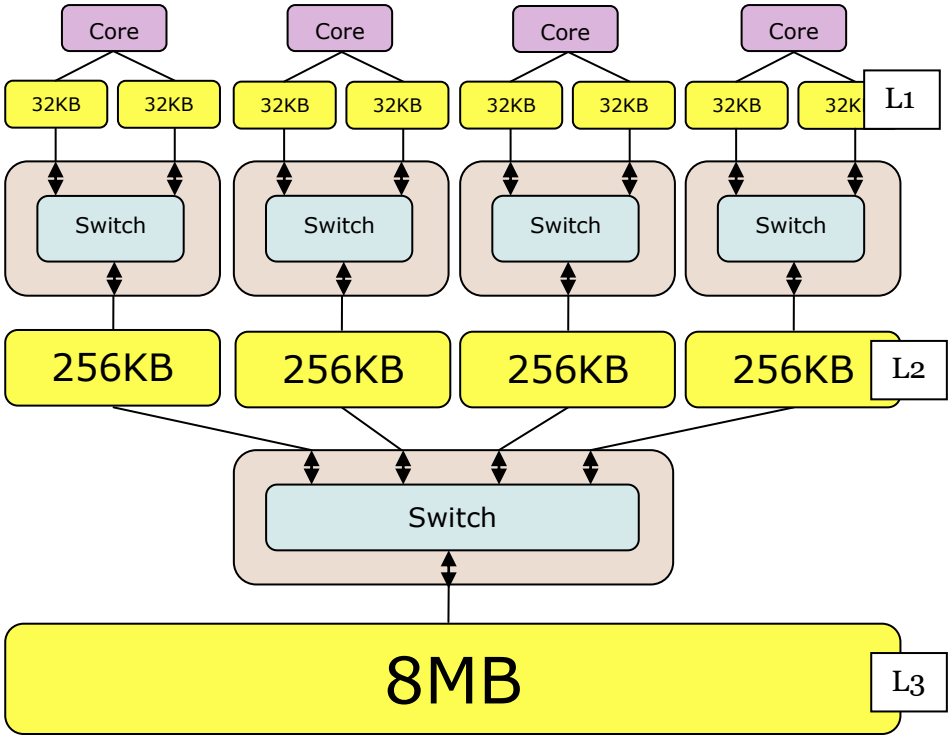

Figura 2.2: Jerarquía de cache en un procesador Intel® Core™ i7-4702EC.

En los procesadores existentes hoy en día en el mercado, usualmente se contemplan tres niveles de memoria cache. Cada núcleo cuenta de forma inherente con su propio primer nivel de jerarquía cache. Este nivel *privado* generalmente está compuesto por sendos módulos independientes para instrucciones y datos (arquitectura *Harvard*) para facilitar su integración en el pipeline. Conviene aclarar que la arquitectura *Harvard* sólo se utiliza en el primer nivel de la jerarquía de cache, por lo que el resto de niveles las caches (al no encontrarse las integradas en el pipeline) son unificadas, ya que esta disposición ofrece mejores prestaciones.

Dependiendo del contexto, el segundo nivel de la jerarquía puede ser también privado o encontrarse compartido entre dos o más núcleos. El tercer nivel, de una forma u otra, es usualmente compartido entre todos los núcleos del procesador.

A continuación se describen algunas de las principales líneas de procesadores actuales, citando la distribución y capacidad de sus jerarquías de cache.

2.2.1 Procesadores Haswell (Intel Corporation)

Haswell es el código de serie de la cuarta generación de procesadores fabricados por Intel Corporation. El chip del procesador tiene capacidad para un máximo de 6 núcleos.

El nivel 1 de cache (L1) se compone de sendos módulos para instrucciones y datos, cada uno dotado con 32KB de capacidad. El nivel 2 (L2) está formado por un módulo unificado de 256KB. Ambos niveles de la jerarquía son privados para cada núcleo.

El nivel 3 (L3) es un único módulo compartido para todos los núcleos del procesador. Tiene una capacidad máxima de 16MB, dependiendo del modelo.

2.2.2 Procesadores Power8 (IBM)

Power8 da nombre a una familia de procesadores de gama alta fabricados por IBM. Diseñados para soportar altas cargas computacionales, su chip puede englobar un máximo de 12 núcleos.

Los niveles de cache 1 y 2 son privados para cada núcleo. El nivel 1 (L1) está diseñado en base a arquitectura Harvard asimétrica, con un módulo de 32KB para instrucciones y otro de 64KB para datos. El nivel 2 (L2) es un módulo unificado de 512KB de capacidad. Ambos niveles se basan en tecnología SRAM.

El nivel 3 (L3) está formado por un único módulo eDRAM de 8MB de capacidad, compartido por todos los núcleos del procesador.

2.3 Resumen

En este capítulo se han descrito los conceptos de memoria y jerarquía de cache, así como la distribución de la cache en los procesadores actuales. Este capítulo ha pretendido aportar una base teórica necesaria para abordar el resto del proyecto.

3. Entorno experimental

Los diferentes ensayos que se han llevado a cabo a lo largo del presente estudio han requerido de la implantación y mantenimiento de un entorno experimental consistente en i) un simulador de procesadores, ii) una *suite* de aplicaciones para evaluar las prestaciones y iii) una infraestructura para el lanzamiento masivo de simulaciones. Todos estos componentes se describen a continuación.

3.1 El simulador Multi2Sim

Multi2Sim es un entorno de simulación de procesadores superescalares, multihilo y multinúcleo de código abierto para sistemas operativos Linux. Permite realizar ejecuciones de aplicaciones en un entorno que simula las prestaciones de computadores reales, con un abanico que abarca desde procesadores de bajas prestaciones con un solo núcleo de ejecución hasta sistemas multinúcleo heterogéneos.

Creado e implementado en el seno del Grupo de Arquitecturas Paralalelas de la Universidad Politécnica de Valencia (UPV) bajo la dirección de Julio Sahuquillo, su desarrollo fue posteriormente liderado por la *Northeastern University* de Boston. Su primera versión lanzada en 2007 estaba orientada exclusivamente para simulación de arquitecturas basadas en MIPS. En la actualidad, colaboran en su mejora y ampliación miembros de varias universidades de todo el mundo, entre los que cabe citar a Salvador Petit y Julio Sahuquillo, del Departamento de Informática de Sistemas y Computadores de la UPV. Hoy en día, Multi2Sim es ampliamente usado por diversos grupos de investigación y fabricantes internacionales como Intel, AMD o NVIDIA.

El simulador, cuya última versión 4.2 se ha empleado en la realización de este trabajo, soporta ahora múltiples arquitecturas (x86, *Evergreen*, etc.) de forma heterogénea, lo que permite realizar simulaciones de sistemas informáticos híbridos, compaginando distintas arquitecturas para CPU y GPU.

Entre sus diversas capacidades de simulación, destacan:

- Entornos superescalares. Emulación de flujos de instrucciones superescalares, modelando características como los etapas *fetch*, *decode*, *issue*, *write-back* y *commit* del ciclo de instrucción, implementación de estructuras tales como *reorder buffers* (ROB), colas para *loads* y *stores*, etc., diferentes tipos de *branch predictions* y ejecución especulativa de instrucciones.

- *Multithreading*. Multi2Sim permite el modelado de procesadores que soportan múltiples hilos de ejecución con diferentes paradigmas multihilo (*simultaneous multi-threading* o SMT, *coarse-grain* y *fine-grain*).

- Multinúcleo. Permite modelar procesadores multinúcleo, donde los diferentes núcleos se comunican entre sí a través de la jerarquía de memoria.

- *Graphics Processing Units*. Multi2Sim dispone de un modelo capaz de simular las últimas GPUs de AMD y NVIDIA y la ejecución de programas OpenCL.

- Jerarquía de Memoria. Soporta múltiples niveles (privados o compartidos entre los núcleos), geometrías de cache y latencias. Incluye una implementación del protocolo MOESI (estados *Modified*, *Owned*, *Exclusive*, *Shared* e *Invalid*) basado en directorios para manejar la coherencia entre las caches de los distintos niveles.

- Redes de interconexión. Los componentes de la jerarquía de memoria se comunican mediante la red de interconexión. Multi2Sim permite seleccionar topologías, anchos de banda, algoritmos de encaminamiento y número de canales virtuales.

A continuación se describen aspectos técnicos y funcionales para el uso y ampliación de las capacidades de Multi2Sim, desde un punto de vista centrado en la simulación de arquitecturas x86. El lector podrá encontrar más información sobre el simulador y sus múltiples posibilidades, así como instrucciones para simulación de otras arquitecturas, en la documentación oficial de Multi2Sim [3].

3.1.1 Código fuente del simulador

Multi2Sim puede descargarse gratuitamente desde el portal oficial www.multi2sim.org. Se distribuye con su código fuente, que ha de compilarse al menos una vez antes de poder ejecutarse. Este código fuente está escrito en lenguaje C, y su licencia de desarrollo GPL (*General Public License*) permite que cualquier usuario pueda adaptarlo a sus necesidades. De hecho, como podrá observarse a lo largo del texto, en ello ha consistido una parte nada desdeñable del trabajo realizado. Aun más, los diferentes experimentos acometidos han hecho necesario el dotar al simulador de algunas características de las que adolecía.

La fig. 3.1.1 describe de forma somera la disposición de los ficheros de código fuente, a fin de dotar al lector de un punto de vista más amplio que le permita abarcar fácilmente las modificaciones desarrolladas, y que serán descritas conforme se aborden los experimentos para los cuales ha sido necesario realizarlas.

- **src**
  Carpeta general de ficheros fuente. Se encuentra en la raíz de Multi2Sim
  - **arch**
    Implementaciones de arquitecturas de microprocesadores.
    Tantos subdirectorios como arquitecturas CPU o GPU soporta el simulador
    - **x86**
      Implementación de la arquitectura x86. Cada arquitectura contiene 3 subdirectorios
      - **asm**
        Desensamblador
      - **emu**
        Simulador funcional
      - **timing**
        Simulador detallado
    - … *otras arquitecturas* …
  - **lib**
    Librerías auxiliares (implementaciones de listas enlazadas, tablas *hash*, *heaps* y otras estructuras de datos)
  - **mem-system**
    Implementación del sistema de memoria: cache, memoria principal, etc.
  - **network**
    Redes de interconexión, con modelos de *switches*, *links*, canales virtuales y algoritmos de enrutamiento
  - **visual**
    Herramienta de visualización basada en GTK
    - **common**
      Archivos comunes: estado, control, etc.
    - **memory**
      Herramienta visual para representar memorias cache y directorios
    - **x86**
      Herramienta visual para representar CPUs multinúcleo de arquitectura x86
    - … *herramientas visuales para otras arquitecturas* …
  - **tools**
    Herramientas adicionales incluidas en Multi2Sim, software de terceros, núcleo OpenCL y OpenGL, etc.

Figura 3.1.1: Estructura del código fuente de Multi2Sim.

3.1.2 Simulación de procesadores con Multi2Sim

Tal y como se ha mencionado en el apartado 3.1.1, Multi2Sim ha de ser compilado antes de poder ser ejecutado. La compilación se realiza desde la carpeta raíz del simulador, mediante el siguiente comando:

```
./configure && make
```

Tras la compilación, en el directorio raíz de Multi2Sim dispondremos ahora de un nuevo subdirectorio llamado *bin*, que contiene el fichero binario *m2s*, que es el fichero ejecutable del simulador.

Las simulaciones en Multi2Sim se lanzan desde la consola de comandos de Linux, lo que permite automatizar las ejecuciones. Para lanzar Multi2Sim basta con invocar al comando *m2s*, seguido de una serie de parámetros que nos permiten indicar información adicional al simulador, tal como número de instrucciones a ejecutar, ficheros de configuración que describen el computador a simular, carga a ejecutar por el sistema simulado o ficheros de salida con los resultados de la simulación, entre otros.

Al tratarse de un fichero ejecutable de Linux, podemos crear un fichero script de *shell* con extensión *.sh* que tenga una línea por cada llamada al ejecutable *m2s*, o bien utilizar el lenguaje propio del shell para iterar entre los distintos ficheros de configuración de memoria, como se muestra en la figura 3.1.2.1, lo que nos permitirá ahorrar tiempo y espacio cuando nos enfrentamos a múltiples simulaciones que difieren en algún parámetro.

```
for mem_config in 512kb_16w 512kb_8w 256kb_16w 256kb_8w
do
        ./m2s --mem-config ${mem_config} …
```

Figura 3.1.2.1: Ejemplo de lanzamiento de múltiples simulaciones con Multi2Sim.

Existe una gran cantidad de parámetros disponibles para ajustar las simulaciones de acuerdo a nuestros objetivos. La figura 3.1.2.2 muestra una línea de ejecución completa, correspondiente a una jerarquía de cache con nivel 2 unifica1do, de 512KB y 16 vías.

```
./m2s --mem-config cacheconfig.4core_l1s_l2u512kb-16w --x86-config
config_multicore --x86-sim detailed --x86-max-cycles 600000000 --
ctx-config ctxconfig.mix2a --mem-report m_512kb-16w.txt --x86-
report x86r_512kb-16w.txt
```

Figura 3.1.2.2: Ejemplo de comando de ejecución para Multi2Sim.

A continuación, se detallan los parámetros utilizados durante los sucesivos experimentos realizados a lo largo del trabajo.

--x86-config *fichero*. Indica que en el archivo con nombre *fichero* se encuentran los parámetros de configuración específicos para arquitectura x86. Para los experimentos realizados, se ha precisado hacer uso de dos secciones:

- [General]
    - ✓ Cores = C. (C = 1 para simular con un solo núcleo, C > 1 para multinúcleo).
    - ✓ FastForward = I. (Nº de instrucciones a ejecutar en modo *FastForward*. Este es un valor global a dividir entre todos los núcleos, por lo que si se quiere realizar "fastforward" de i instrucciones en cada núcleo es necesario especificar un valor de $I = i \cdot C$, siendo C el número de núcleos del computador).

- [Queues]

    - ✓ RobSize = M. (Tamaño del *reorder buffer*, expresado en microinstrucciones).
    - ✓ LsqSize = M. (Tamaño de la cola *load-store*, en microinstrucciones).
    - ✓ RfIntSize = M. (Número de registros físicos para datos enteros).
    - ✓ RfFpSize = M. (Número de registros físicos para coma flotante).
    - ✓ RfXmmSize = M. (Número de registros físicos MMX – instrucciones SIMD –).

Todos los parámetros son opcionales. En caso de omisión, el simulador adopta valores por defecto. Por ejemplo, omitiendo el parámetro Cores, se definirá un computador con un solo núcleo y un solo hilo de ejecución.

--mem-config *fichero*. Indica que en el archivo con nombre *fichero* se encuentran los parámetros de configuración específicos para la jerarquía de memoria.

Este fichero de configuración puede tener cualquier extensión, por lo que para diferenciarlos a lo largo de los experimentos se ha seguido una nomenclatura consistente en denominar a todos estos ficheros con el término *cacheconfig*, indicativo de su utilidad como fichero de configuración de jerarquía de cache, y una extensión que haga alusión a la propia disposición de la jerarquía; así, el fichero con nombre *cacheconfig.l1h_l2u1mb1024s16w* será indicativo de una cache L1 Harvard y una cache L2 unificada de 1MB, 1024 conjuntos (*sets*) y 16 vías (*ways*).

La figura 3.1.2.3 muestra el contenido del fichero cacheconfig.l1h_l2u1mb1024s16w, que ha sido empleado en el primero de los experimentos (véase apartado 4.1). En la figura pueden apreciarse las diversas secciones en que se compone un fichero de configuración de jerarquía de memoria en Multi2Sim. Algunas de ellas son fácilmente identificables.

Comenzamos definiendo, con las entradas *CacheGeometry*, sendas geometrías de memoria cache, a las que denominamos –de forma arbitraria–, *l1topo* y *l2topo*. Cada geometría se define en base a una serie de parámetros: número de conjuntos (*Sets*), tamaño de bloque en Bytes (*BlockSize*), latencia expresada en ciclos (*Latency*), número de vías (*Assoc*, del inglés *Associativity*) y política de reemplazo (Policy). Multi2Sim permite variar otros muchos parámetros de la memoria, sin bien para un experimento inicial bastarían con los aquí reseñados. El tamaño de la memoria, en Bytes, puede obtenerse mediante el producto del tamaño del bloque, número de vías y número de conjuntos, por lo que la topología *l1topo* define una memoria cache de 64B * 2 * 128 = 16KB, y la topología *l2topo* define a su vez una memoria cache de 64B * 16 * 1024 = 1MB.

Las siguientes dos entradas, tituladas *Network*, definen sendas redes de interconexión (o buses). Ambas tienen un ancho de banda (*DefaultBandwidth*) de 32B, así como búferes de entrada (*DefaultInputBufferSize*) y de salida (*DefaultOutputBufferSize*) de un tamaño cuatro veces mayor. Las redes nos permitirán unir entre sí los diferentes elementos de la jerarquía de memoria: procesador, memorias cache y memoria principal.

```
[ CacheGeometry l1topo ]
Sets = 128
Assoc = 2
BlockSize = 64
Latency = 2
MSHR = 4
Policy = LRU

[ CacheGeometry l2topo ]
Sets = 1024
Assoc = 16
BlockSize = 64
Latency = 6
MSHR = 4
Policy = LRU

[ Network net-0 ]
DefaultInputBufferSize = 128
DefaultOutputBufferSize = 128
DefaultBandwidth = 32

[ Network net-1 ]
DefaultInputBufferSize = 128
DefaultOutputBufferSize = 128
DefaultBandwidth = 32

[ Module dl1 ]
Type = Cache
Geometry = l1topo
LowNetwork = net-0
LowModules = l2

[ Module il1 ]
Type = Cache
Geometry = l1topo
LowNetwork = net-0
LowModules = l2

[ Module l2 ]
Type = Cache
Geometry = l2topo
HighNetwork = net-0
LowNetwork = net-1
LowModules = mod-mm

[ Module mod-mm ]
Type = MainMemory
HighNetwork = net-1
BlockSize = 64
Latency = 100

[ Entry core-0 ]
Arch = x86
Core = 0
Thread = 0
DataModule = dl1
InstModule = il1
```

Figura 3.1.2.3: Contenido del fichero *cacheconfig.l1h_l2u1mb1024s16w*

Las entradas *Module*, definen los distintos módulos físicos de memoria, esto es, implementaciones de las distintas topologías lógicas definidas con *CacheGeometry* o de la memoria principal. Así, una misma geometría puede utilizarse en varias implementaciones, lo que supone un importante ahorro a la hora de definir jerarquías complejas de memoria. Para nuestros experimentos se han definido cuatro módulos. Los dos primeros son implementaciones de la geometría *l1topo* denominadas *dl1* e *il1*. Como el lector habrá sin duda podido adivinar, serán los módulos de cache de nivel 1 de datos e instrucciones, respectivamente. El segundo, implementación de *l2topo* y denominado *l2*, será la memoria cache unificada de nivel 2.

Como podemos observar, la definición de un módulo se realiza de una forma muy simple. Basta con expresar su tipo (*Type*), que en ambos casos tendrá un valor igual a "Cache", la geometría (*Geometry*) que implementa, y las redes de interconexión que utilizará para comunicar con el nivel inferior de la jerarquía de memoria (*LowNetwork*) y con el nivel superior (*HighNetwork*). Nótese que el procesador se encuentra en el nivel más alto de la jerarquía, y la memoria principal en el nivel más bajo de todos.

El parámetro MSHR (*Miss Status Holding Register*) indica el número de entradas del registro de fallos de accesos a cache soportados simultáneamente. El valor por omisión de Multi2Sim es de 16 entradas.

El cuarto módulo, *mod-mm*, no implementa una geometría de cache, sino que define la propia memoria principal del sistema. Por tanto, el parámetro *Type* tiene asociado un valor igual a "MainMemory", y no se declara *LowNetwork*, ya que no existirán módulos de nivel inferior. Al no poder leerse desde una geometría lógica predefinida, han de indicarse de forma explícita parámetros como el tamaño del bloque y la latencia.

Por último, la sección *Entry* define un núcleo de ejecución o núcleo. Multi2Sim permite expresar multitud de parámetros para definir el procesador, si bien optaremos por dejar la mayor parte de ellos a sus valores por defecto, indicando solamente su arquitectura (*Arch*), que estableceremos a "x86", los módulos de memoria cache de instrucciones y datos que lleva asociado (*InstModule* y *DataModule*) y el número de procesador e hilo (Núcleo y *Thread*). Definimos así una CPU monohilo cuya frecuencia de reloj será de 1000MHz (el valor por defecto del simulador).

--ctx-config *fichero*. Indica que en el archivo con nombre *fichero* se encuentran los parámetros de configuración de los contextos de ejecución. Los contextos especifican la carga de trabajo o aplicaciones que el computador modelado ejecutará. En simulaciones de computadores multinúcleo cada contexto puede asignarse a varios hilos en los distintos núcleos, lo que permite la ejecución de cargas paralelas, multiprogramadas e híbridas.

En la figura 3.1.2.4 se muestra un ejemplo de fichero de contexto, correspondiente al programa *gromacs*. Como puede observarse en la figura anterior, el fichero de contexto comienza por una entrada *[Context]* que incluye el número de contexto.

```
[ Context 0 ]
exe = gromacs_base.i386
args = -silent -deffnm gromacs -nice 0
cwd =  /spec2006-x86-bin/435.gromacs
StdOut = gromacs.out
```

Figura 3.1.2.4: Ejemplo de fichero de contexto con un solo núcleo para *gromacs*.

Los parámetros de los ficheros de contexto se describen a continuación:

- exe: indica el fichero binario ejecutable (en este caso, para arquitectura x86, extensión i386)
- args: argumentos que acepta el programa ejecutable
- cwd: directorio de trabajo. Ruta en la que se encuentra el fichero binario
- StdOut: fichero de salida estándar con los resultados de la ejecución del programa (nótese que se trata de la salida de la ejecución del programa simulado, no del simulador en sí)

Al modelar procesadores con varios núcleos, pueden ejecutarse cargas de trabajo diferentes en cada uno de los núcleos, para lo cual basta con crear sendas entradas *Context* en el fichero que se pasa en el parámetro --ctx-config, donde cada uno tendrá los ficheros asociados al *benchmark* que ejecutará el núcleo en cuestión

Por defecto, Multi2Sim vuelca el informe resumen de salida por pantalla, por lo que puede redirigirse la salida a un fichero por medio de una tubería Linux, para el posterior tratamiento estadístico de los datos obtenidos. La figura 3.1.2.5 muestra un ejemplo de resumen de ejecución, correspondiente a la simulación de un computador de 4 núcleos.

```
[ General ]
RealTime = 10921.96 [s]
SimEnd = x86MaxCycles
SimTime = 600000349.00 [ns]
Frequency = 1000 [MHz]
Cycles = 600000350

[ x86 ]
RealTime = 10921.95 [s]
Instructions = 3702292144
InstructionsPerSecond = 338977
Contexts = 4
Memory = 53161984
FastForwardInstructions = 2000000000
CommittedInstructions = 829251455
CommittedInstructionsPerCycle = 1.382
CommittedMicroInstructions = 1475028177
CommittedMicroInstructionsPerCycle = 2.458
BranchPredictionAccuracy = 0.9166
SimTime = 600000000.00 [ns]
Frequency = 1000 [MHz]
Cycles = 600000000
CyclesPerSecond = 54935
```

Figura 3.1.2.5: Ejemplo del informe de salida de una ejecución de Multi2Sim.

--x86-max-cycles *valor*. Indica en *valor* el número máximo de ciclos a ejecutar por cada núcleo del computador simulado. En lugar de limitar la ejecución por número de ciclos, puede restringirse por número de instrucciones, utilizando el parámetro *--x86-max-inst* de forma análoga.

--mem-report *fichero*. Vuelca en el archivo con nombre *fichero* el informe de salida de la jerarquía de memoria. Para cada módulo de cache y la memoria principal se recogen multitud de datos, entre los que cabe destacar accesos, aciertos, fallos, tasas de acierto, lecturas, escrituras, fallos de lectura y escritura, escrituras bloqueantes y no bloqueantes, etc.

--x86-report *fichero*. Vuelca en el archivo con nombre *fichero* el informe de salida de la arquitectura x86. Al igual que el informe de la jerarquía de memoria, incluye gran cantidad de datos: tiempo de ejecución, uso de memoria, cantidad y tipo de

instrucciones ejecutadas así como estadísticas de estado de instrucciones, estadísticas de predicción de saltos, estado de las colas, etc.

3.2 SPEC CPU2006

The Standard Perfomance Evaluation Corporation (SPEC, por sus siglas en inglés), es una organización sin ánimo de lucro con sede en Gainesville, Virginia, Estados Unidos de América, cuyos objetivos son el diseño, mantenimiento y difusión de un conjunto estandarizado de *benchmarks* y métricas para la evaluación del rendimiento de los sistemas informáticos actuales. Fundada en 1988 gracias a la cooperación de un pequeño grupo de fabricantes de computadores, en la actualidad cuenta con más de 60 empresas tecnológicas de ámbito mundial.

SPEC CPU2006 es la versión actualmente en uso del conjunto de *benchmarks* SPEC CPU. Lanzada el 6 de agosto de 2006 en sustitución de la anterior CPU2000, su última actualización, codificada como V1.2, data de septiembre de 2011.

SPEC CPU2006 compone en la actualidad una de las más importantes herramientas de evaluación de sistemas informáticos tanto en la industria como en el ámbito académico. Tasa el rendimiento conjunto del procesador (CPU), la arquitectura de memoria y los compiladores de lenguajes de programación. Se compone de dos grandes subgrupos de *benchmarks*: SPECint2006 (también conocido como CINT2006), formado por 12 aplicaciones de aritmética entera, y SPECfp2006 (ó CFP2006), integrado por 17 aplicaciones que hacen uso de aritmética en coma flotante. Estos 29 programas de evaluación se distribuyen con código fuente y han de compilarse antes de su ejecución.

Todas las cargas ejecutadas durante los experimentos realizados han consistido en diferentes *benchmarks* de SPEC CPU2006. Para ello, se han compilado previamente los núcleos de SPEC CPU a fin de obtener programas ejecutables en arquitectura x86.

3.2.1 Descripción de los *benchmarks*

A continuación se incluye una breve descripción de cada uno de los *benchmarks* que componen la *suite* SPEC CPU2006. Para cada *benchmark* se incluye el grupo al que pertenece –aritmética entera o en coma flotante–, así como su categoría funcional, dada la naturaleza del programa real que ejecuta.

La información aquí mostrada procede de *SPEC CPU2006 Benchmark Description* [4].

400.perlbench. Grupo: aritmética entera. Categoría: lenguaje de programación. Desarrollado en ANSI C, se trata de una versión reducida del intérprete del lenguaje Perl. Su carga de trabajo la componen tres *scripts*, que emulan el programa *antiSPAM* de código abierto *SpamAssassin*, el convertidor *mail*/HTML *MhonArc* y el *specdiff* de las *SPECtools* (todos ellos escritos en Perl).

401.bzip2. Grupo: aritmética entera. Categoría: compresor. Desarrollado también en ANSI C, está basado en el popular compresor de ficheros de código abierto *bzip2* y realiza diferentes tareas de compresión y descompresión de archivos. Para su ejecución

utiliza una carga de trabajo compuesta por dos pequeñas imágenes JPEG, un programa binario, un fichero TAR con código fuente, un archivo HTML y un fichero combinado.

403.gcc. Grupo: aritmética entera. Categoría: compilador. Se trata de una alteración de la versión 3.2 del compilador GCC, por lo que huelga decir que ha sido desarrollado en C. Su ejecución realiza la compilación de una carga de trabajo compuesta de nueve programas C preprocesados (extensión .i) activando gran parte de los *flags* de optimización de compilación existentes en GCC. Como curiosidad, el código resultado de la ejecución del programa se compone de ficheros ensamblador compilados para un procesador *AMD Opteron*.

429.mcf. Grupo: aritmética entera. Categoría: optimización combinacional. Desarrollado en ANSI C con un uso extensivo de sus librerías matemáticas, es un programa basado en la aplicación MCF, usada para la programación de rutas de transporte público, y que realiza estimaciones de tiempos de trayectos basadas en las rutas y horarios disponibles.

445.gobmk. Grupo: aritmética entera. Categoría: juegos, inteligencia artificial. Desarrollado en C nativo, simula una partida de *GO*, un juego de mesa tradicional chino de fuerte componente estratégico, donde dos oponentes mueven alternativamente fichas negras y blancas en un tablero de 19 filas por 19 columnas. Su carga de referencia es un fichero en formato *SmartGo* (.sgf), que se ha hecho popular en los países de Extremo Oriente para almacenar en ASCII secuencias de movimientos de partidas de *GO*.

456.hmmer. Grupo: aritmética entera. Categoría: búsqueda en bases de datos de genoma, inteligencia artificial. Este programa escrito en C utiliza modelos ocultos de Markov (HMMs), técnica de inteligencia artificial, para realizar búsquedas rápidas basadas en reconocimiento de patrones (secuencias de ADN) en una base de datos de cadenas genéticas. Como carga de trabajo utiliza una base de datos y una serie de modelos ocultos de Markov, realizando búsquedas aleatorias de diferentes cadenas de ADN, y devolviendo un fichero con las coincidencias (*matchings*).

458.sjeng. Grupo: aritmética entera. Categoría: inteligencia artificial, búsqueda en árbol, reconocimiento de patrones. Es un simulador de ajedrez escrito en ANSI C y basado en la versión 11.2 del programa Sjeng. Carga un fichero de texto que contiene nueve partidas de ajedrez en distintas fases de desarrollo escritas en la notación estándar FEN (*ForsythEdwards Notation*) y calcula las mejores jugadas posibles para cada una de las distintas situaciones.

462.libquantum. Grupo: aritmética entera. Categoría: física, computación cuántica. Escrito en lenguaje ISO/IEC 9899 (conocido como C99), simula un computador cuántico, máquinas basadas en el concepto de *qbit* o bit cuántico, que por su ínfimo tamaño se encuentran sujetas a las leyes de la mecánica cuántica y son capaces de resolver problemas de gran complejidad en un tiempo polinomial. *Libquantum* proporciona herramientas para simular registros cuánticos y algunas puertas lógicas elementales y realiza la factorización de un número que se le proporciona como entrada.

464.h264ref. Grupo: aritmética entera. Categoría: compresión de vídeo. Es una implementación de referencia del estándar de compresión de vídeo H264/AVC (*Advanced Video Codin*g). Escrito en C, su carga de trabajo la componen dos ficheros de entrada sin comprimir en formato YUV, un formato utilizado para compresión de vídeo (120 *frames* a 176x144 píxeles) y SSS, utilizado para cinemáticas de videojuegos (171 *frames* a 512x320 píxeles).

471.omnetpp. Grupo: aritmética entera. Categoría: simulación de eventos, redes. Simula una gran red Ethernet (*Campus backbone*) mediante el conocido programa OMNeT++. La red representada está disponible de forma pública y comprende cerca de 8000 *hosts* y 900 *switches* y *hubs*, tanto *Fast Ethernet* como *Gigabit Ethernet*, en modos *half* y *full duplex*. Como carga de trabajo recibe un fichero en lenguaje de descripción de redes de OMNeT++ (extensión .ned) que contiene la topología completa de la red. El programa, al igual que el simulador original, está escrito en C++.

473.astar. Grupo: aritmética entera. Categoría: juegos, inteligencia artificial. Desarrollado en C++, deriva de una librería de búsqueda de caminos 2D usada en videojuegos. Implementa tres versiones distintas del algoritmo de búsqueda A* (*A-estrella*, de ahí el nombre del núcleo): la primera y más simple, para mapas que contienen zonas permitidas y prohibidas, una modificación de esta para mapas que además contengan zonas de distinta velocidad de movimiento, y una tercera para búsqueda avanzada en grafos. Su carga de trabajo es un mapa en formato binario, y calcula todos los caminos posibles dentro de él.

483.xalancbmk. Grupo: aritmética entera. Categoría: transformación de documentos XML. Es una versión modificada de XalanC++, un potente procesador XSLT que sigue las normas del W3C (*Wold Wide Web Consortium*) para la transformación XSL, convirtiendo documentos en formato XML a HTML, texto plano u otros tipos de ficheros XML. Sus datos de entrada son un documento XML y una hoja de estilos XSL (formato específico para XML). Devuelve como resultado de su ejecución un documento HTML bien formado.

410.bwaves. Grupo: coma flotante. Categoría: física, dinámica de fluidos. Desarrollado en Fortran77, realiza la simulación numérica de ondas de choque en un fluido laminar tridimensional, ofreciendo una implementación del llamado algoritmo *BiCGstab*, que resuelve sistemas asimétricos de ecuaciones no lineales de forma iterativa. Como entrada recibe parámetros numéricos como el tamaño de la malla que contiene el fluido o las especificaciones de la dinámica de flujo que imperará en el sistema simulado.

416.gamess. Grupo: coma flotante. Categoría: química cuántica. Escrito en Fortran, se trata de un programa que realiza un amplio abanico de cálculos cuánticos sobre simulaciones de moléculas de complejidad creciente, desde combinaciones inorgánicas simples como el agua, hasta compuestos orgánicos como la citosina, una de las bases nitrogenadas que componen el ADN.

433.milc. Grupo: coma flotante. Categoría: física, cromo-dinámica cuántica. Este programa escrito en C realiza simulaciones en cuatro dimensiones de la teoría de Retículos aplicada a computadores paralelos MIMD (*Multiple Instruction, Multiple Data*). Como curiosidad, el código de que se compone *milc* es ampliamente usado en

centros de investigación y supercomputadores, dado su valor para la resolución de problemas complejos.

434.zeusmp. Grupo: coma flotante. Categoría: física, magneto-hidrodinámica. Escrito en Fortran y Fortran77, se basa en el código de ZEUSMP, un simulador de dinámica de fluidos desarrollado para estudiar fenómenos astrofísicos tales como campos gravitacionales, mediante la resolución de problemas de hidrodinámica y magneto-dinámica no relativistas. Su carga de trabajo es un único fichero *zmp_inp*, que contiene información sobre constantes físicas, ecuaciones de estado del sistema y datos de control.

435.gromacs. Grupo: coma flotante. Categoría: química, dinámica molecular. Es una versión reducida (en C y Fortran) de GROMACS, un programa que simula dinámicas moleculares tales como las ecuaciones fundamentales del movimiento de Newton para poblaciones de cientos de millones de partículas. Devuelve como salida la temperatura media del sistema analizado y el número de operaciones de coma flotante ejecutadas.

436.cactusADM. Grupo: coma flotante. Categoría: física, relatividad general. Es una combinación de Cactus, un entorno de resolución de problemas de código abierto, y BenchADM, el núcleo computacional de muchas aplicaciones de análisis de relatividad general. Escrito en Fortran90 y ANSI C, este *benchmark* calcula la solución de las ecuaciones de evolución de Einstein, que describen cómo se curva el espacio/tiempo como consecuencia de su contenido en masa y energía.

437.leslie3d. Grupo: coma flotante. Categoría: física, dinámica de fluidos. Desarrollado en Fortran90, deriva de LESlie3d, un simulador de dinámica de fluidos computacional usado en la investigación de una amplia gama de fenómenos de combustión, acústicos, y aplicaciones generales de la mecánica de fluidos. Concretamente, la carga de referencia simula corrientes de flujo en un proceso de combustión.

444.namd. Grupo: coma flotante. Categoría: biología estructural. Desarrollado en C++, es una modificación de NAMD, un programa paralelo de simulación de grandes sistemas biomoleculares. El modelo de carga del *benchmark* lo forma un sistema poli proteínico compuesto de casi cien mil átomos, devolviendo como salida varias sumas de comprobación de los cálculos realizados.

447.dealII. Grupo: coma flotante. Categoría: matemáticas, ecuaciones diferenciales. Este *benchmark* usa deal.II, un programa que utiliza las herramientas matemáticas más avanzadas de C++ para desarrollar modelos algorítmicos finitos. En particular, el modelo de carga resuelve una ecuación de tipo *Helmholtz* con coeficientes variables mediante refinamientos sucesivos. La salida describe el estado del problema así como el grado de refinamiento de la solución alcanzada.

450.soplex. Grupo: coma flotante: Categoría: matemáticas, programación lineal. Escrito en ANSI C++, este *benchmark* resuelve un programa lineal (conjunto de inecuaciones lineales) mediante el algoritmo *Símplex*. El sistema de inecuaciones se trata como una matriz de m filas y n columnas, y se emplea factorización LU para su resolución. La ejecución devuelve el vector solución (de tamaño N) y el número de iteraciones realizadas.

453.povray. Grupo: coma flotante. Categoría: visión por computador. POVRay, desarrollado en ISO C++, renderiza imágenes tridimensionales a través de la técnica de trazado de rayos (*raytracing*), basada en la incidencia de los rayos de luz sobre un cuerpo sólido, simulando la reflexión y refracción producidas respectivamente por superficies opacas y transparentes. El *benchmark* renderiza la imagen con *antialising* de un tablero de ajedrez tridimensional de 1280x1024 píxeles, devolviendo la imagen construida y un log con información estadística sobre las operaciones ejecutadas.

454.calculix. Grupo: coma flotante. Categoría: física, mecánica estructural. Basado en *CalculiX* y escrito en Fortran y C, este programa realiza cálculos de estructuras tridimensionales, sean estáticas (puentes y edificios) o dinámicas (análisis de roturas, resistencia a terremotos...). El modelo de carga del *benchmark* simula la deformación de un disco compresor debido a la acción de la fuerza centrífuga. Su salida consiste en información sobre el estado final de la estructura y el desplazamiento sufrido por la misma durante la simulación.

459.GemsFDTD. Grupo: coma flotante. Categoría: física, electromagnética computacional. Desarrollado en Fortran90, resuelve las ecuaciones de Maxwell que definen los fenómenos del campo electromagnético en el espacio, mediante el Método de las Diferencias Finitas en el Dominio del Tiempo (*FDTD*, por sus siglas en inglés). Los parámetros de entrada del *benchmark* definen el tamaño del problema y las ecuaciones de estado del campo electromagnético.

465.tonto. Grupo: coma flotante. Categoría: cristalografía cuántica. Escrito en Fortran95, realiza simulaciones de estructuras cristalinas cuánticas a partir de datos de entrada tales como la disposición de los átomos y datos de refracción de rayos X en la superficie del cristal.

470.lbm. Grupo: coma flotante. Categoría: física, dinámica de fluidos computacional. Es una implementación en ANSI C del Método *Lattice Boltzmann* (LBM) para la simulación tridimensional de fluidos irreales y conforma el núcleo de diversas aplicaciones usadas en centros de investigación. La carga de referencia realiza tres mil iteraciones de simulación, devolviendo datos sobre el estado del fluido simulado en forma de vectores de tres componentes.

481.wrf. Grupo: coma flotante: Categoría: predicción climática. Desarrollado en Fortran90 y C, está basado en el modelo *Weather Research and Forecasting* (WRF), un sistema de predicción de las condiciones atmosféricas de última generación. La carga de referencia realiza la predicción para un periodo de 24 horas a intervalos de 3, en un área dada de 10 km de diámetro, devolviendo las diferentes zonas de temperatura estimada para cada uno de los pasos realizados.

482.sphinx3. Grupo: coma flotante. Categoría: reconocimiento de texto. *Sphinx3* es un programa de reconocimiento escrito en C que realiza la transcripción de audio en formato RAW a caracteres ASCII. Los ficheros usados en la carga de trabajo proceden de la base de datos CMU o AN4, grabada por la *Carnegie Mellon University* en 1991 y que ha servido de base a multitud de tesis y artículos de investigación.

3.3 Infraestructura de simulación

Por su naturaleza, los experimentos realizados en este trabajo precisan de una infraestructura capaz de dar soporte a los requerimientos de Multi2Sim en simulaciones de sistemas multinúcleo.

A tal fin, el grupo de investigación GAP de la Universidad Politécnica de Valencia ha proporcionado acceso a uno de sus clústeres de computadores. Este clúster, utilizado por investigadores, doctorandos y personal colaborador del GAP, cuenta con la suficiente capacidad de cómputo para obtener los resultados de las simulaciones en cuestión de horas de ejecución, en función de la tasa de ocupación de sus nodos.

3.3.1 Arquitectura hardware

Para la realización de los diferentes experimentos, el simulador Multi2Sim se ha instalado y ejecutado desde el clúster ***eri1***. El clúster lo forman 18 nodos en formato *blade* cada uno con:

- 2 *hexacore* Intel Xeon de 2.4GHz
- 48GB DDR3 1333MHz de memoria RAM
- 1 disco duro SATA de 256GB

Por tanto, se ha contado con una capacidad de cómputo total de 18 x 2 x 6 = 216 núcleos y 864GB de memoria RAM.

Dado que el clúster eri1 es un recurso compartido entre múltiples usuarios, se le ha dotado de un sistema de gestión de trabajos denominado *Condor*, descrito en el apartado siguiente. Todas las operaciones y ejecuciones en el clúster se han realizado desde consola de comandos, mediante acceso SSH seguro al *frontend* del sistema.

3.3.2 El sistema *Condor*

Condor es un sistema de gestión de cargas para tareas de computación intensiva, de código abierto, software producto del *Condor Research Project* y desarrollado por el Departamento de Ciencias de la Computación de la Universidad de Wisconsin, en Madison, Estados Unidos de América.

Ampliamente utilizado en la actualidad tanto a nivel industrial como en el ámbito científico, *Condor* constituye un sistema eficiente de administración y monitorización que provee un mecanismo de manejo de colas, políticas de planificación, esquema de prioridades y gestión de recursos.

La ejecución, por parte de un usuario, de un trabajo en un computador o clúster gestionado por *Condor*, implica la creación de una petición al sistema que es colocada

en una cola, donde a través de un proceso de selección se establece dónde (en qué nodo) y cuándo se ejecutará.

Condor incluye múltiples funcionalidades, entre las que cabe destacar:

- Directiva *ClassAds*: Proporciona un marco de trabajo flexible y expresivo para evaluar si las solicitudes de acceso a recursos pueden ser satisfechas en base a los recursos ofrecidos por el sistema.

- Entrega distribuida: no existe un nodo central que reciba y entregue las tareas encoladas. En su lugar, estas se distribuyen directamente en varios nodos, los cuales disponen de colas de trabajos independientes.

- Gestión de prioridad de usuario: *Condor* permite asignar prioridades a los usuarios en base a múltiples factores (compartición justa, penalización para los usuarios con mayor cantidad de tareas enviadas, asignación de prioridad estricta *ad hoc*, etc.). La gestión recae en los administradores del sistema.

- Gestión de prioridad de tareas: de forma análoga a la anterior, las tareas *de un mismo usuario* pueden disponer de distintos niveles de prioridad. A diferencia de la anterior, cada usuario puede gestionar el orden de prioridad de sus propias tareas.

- Soporte para tareas simultáneas: permite gestionar tareas serie y paralelas PVM y MPI.

- Puntos de verificación –*checkpoints*–: *Condor* permite crear puntos de verificación de forma transparente al usuario. Un *checkpoint* es una instantánea del estado de una tarea en un instante dado. La tarea puede reanudar su ejecución desde el punto de verificación en cualquier momento, lo que permite la migración transparente de tareas entre los nodos del sistema de forma que aparentemente estas mantengan una ejecución ininterrumpida. El sistema de puntos de verificación proporciona una herramienta de tolerancia a fallos, que garantiza la utilización del tiempo de ejecución acumulado para una tarea, minimizando la penalización de posibles fallos hardware que inhabiliten parcialmente el sistema.

- Suspensión y reanudación de tareas: *Condor* gestiona directamente con el Sistema Operativo la suspensión y reanudación de tareas.

- Plataformas heterogéneas: soporte para sistemas Linux, UNIX y Windows.

- Autenticación y autorización: *Condor* soporta diversos mecanismos de autenticación de red como Kerberos, lo que permite incluir certificados digitales basados en clave pública.

- *Grid computing*: *Condor* incorpora funcionalidades basadas en computación en Grid. Permite recibir tareas de otros clústeres, computadores y sistemas

distribuidos a través del *toolkit Globus*. *Condor* puede también distribuir tareas a recursos administrados por otros sistemas de planificación (como PBS) a través de *Globus*.

Algunos comandos importantes

Condor dispone de una serie de órdenes de consola que permiten al usuario realizar operaciones tales como programar, eliminar o consultar el estado de las tareas, asignar prioridades, etc. A continuación se detalla un breve resumen de las que han sido empleadas durante los diferentes experimentos.

condor_run "comando". Envía una línea de comandos Shell como trabajo a ejecutar en *Condor*. Esta orden espera a que el trabajo finalice, escribe la salida en el terminal de consola, y termina con el código de estado que haya devuelto el trabajo de *Condor*. No se muestra salida alguna por pantalla hasta que el trabajo haya finalizado (de forma correcta o con error).

condor_q. Muestra por pantalla un listado de los trabajos existentes en la cola de *Condor*. Para cada trabajo se muestra su identificador interno unívoco, el usuario que lo ha creado, su estado (R –*running*, en ejecución–, I –*idle*, en pausa– o H –*held*, retenido–), la fecha y hora de lanzamiento, el tiempo de ejecución acumulado en el sistema, la memoria ocupada y la orden *Shell* que contiene. Con el parámetro *submitter usuario* pueden filtrarse los resultados para un usuario dado.

condor_rm. Elimina uno o más trabajos de la cola de *Condor*. La eliminación no es instantánea, si no que ha de procesarse y validarse antes por el sistema. Si se ejecuta esta orden seguida de un número, marca para borrado el trabajo cuyo id coincida con el suministrado. Puede utilizarse *condor_rm user usuario* para eliminar todos los trabajos del usuario cuyo nombre se pase por parámetro.

condor_status. Muestra información detallada de cada máquina / nodo del sistema: capacidad, estado (libre, ocupado). Útil para monitorizar el estado del sistema.

condor_prio –p prioridad idtrabajo. Permite establecer prioridades separadas para cada uno de los trabajos lanzados *por el usuario actual*. A menor valor, mayor prioridad de ejecución tendrá el trabajo en cuestión. Por defecto, todos los trabajos del mismo usuario se envían a la cola con la misma prioridad. Nótese que el sistema de colas establece dos niveles de prioridades: de trabajo y de usuario, prevaleciendo este último (todos los trabajos de un usuario A tendrán preferencia de ejecución sobre los de otro usuario B si la prioridad de usuario de A es mayor que la de B, aunque B asignase internamente a sus trabajos prioridades superiores a las que A estableciese para los suyos). Las prioridades de usuario fluctúan con el tiempo –salvo contraorden del administrador del sistema–, por lo que los usuarios verán disminuir su prioridad conforme aumente la carga de trabajos que envían al sistema, de forma que se establezca un sistema justo de asignación de recursos. Las prioridades actuales a nivel de usuario pueden consultarse con el comando *condor_userprio*.

3.4 Resumen

A lo largo de este capítulo, se ha descrito de forma extensiva el entorno experimental en que se han realizado las distintas pruebas del presente trabajo.

Comenzando por el simulador Multi2Sim, se ha realizado un análisis de sus capacidades y la estructura de su código fuente, así como pormenorizado los parámetros y valores utilizados para su configuración y el modelado de procesadores.

En el apartado correspondiente a SPEC CPU2006, se han descrito los diferentes *benchmarks* que componen las cargas de trabajo seleccionadas para las ejecuciones de los procesadores simulados en Multi2Sim.

Por último, se describe la infraestructura de simulación utilizada para el lanzamiento de los experimentos: la arquitectura del clúster utilizado y el sistema de gestión de colas de trabajos que asegura un uso equitativo de los recursos *hardware*.

4. Resultados experimentales

Este capítulo describe las sucesivas pruebas realizadas, así como el análisis de los datos obtenidos en ellas. Comenzaremos detallando la naturaleza de los procesadores modelados, para describir después los experimentos ordenados de forma cronológica.

Se ha trabajado con dos arquitecturas diferentes de chip, la primera de ellas está formada por un procesador con un solo núcleo, memoria cache de nivel 1 *Harvard* y cache de nivel 2 unificada, como se muestra en la figura 4.1.

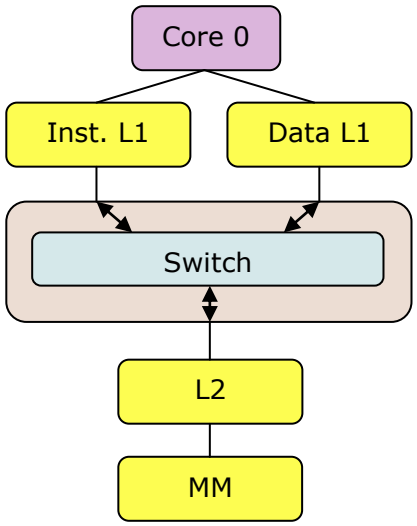

Figura 4.1: Esquema del procesador con un solo núcleo.

La segunda arquitectura (ver figura 4.2) consiste en un procesador de cuatro núcleos, donde cada uno cuenta con su propio par de módulos de memoria cache de nivel 1, existiendo un único módulo de memoria cache de nivel 2 compartido.

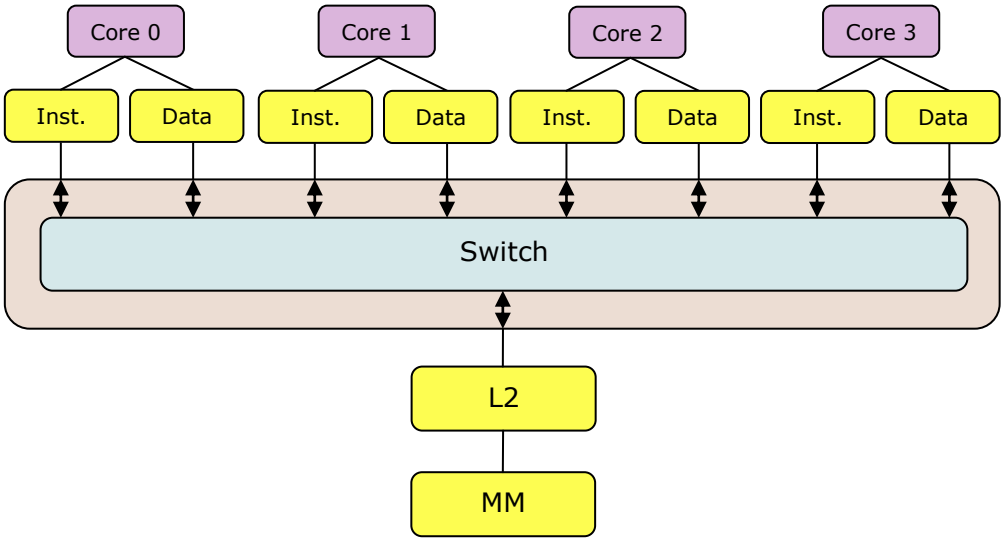

Figura 4.2: Esquema del procesador de cuatro núcleos.

El cuadro 4.3 resume los valores de configuración del simulador comunes a todos los experimentos realizados.

1) Sistema de memoria

- Cache de nivel 1
  - Tipo: Harvard
  - Tamaño: 16KB
  - Asociativa por conjuntos de 2 vías
  - Tamaño de bloque: 64B
  - Latencia: 2 ciclos
  - Política: LRU
  - MSHR: 4
- Cache de nivel 2
  - Tipo: Unificada
  - Tamaño de bloque: 64B
  - Latencia: 6 ciclos
  - Política LRU
  - MSHR: 4
- Memoria Principal
  - Latencia: 300 ciclos

2) Arquitectura x86

- FastForward: 500M instrucciones por núcleo
- Colas y registros
  - RobSize: 256 microinstrucciones
  - LsqSize: 256 microinstrucciones
  - RfIntSize: 300 registros
  - RfFpSize: 300 registros
  - RfXmmSize: 300 registros

3) Otros parámetros

- Límite por ejecución: 600M ciclos
- Clase de simulación (nivel de información): Detallada

Cuadro 4.3: Parámetros fijos de las simulaciones con Multi2Sim.

La primera sección corresponde a los parámetros de la jerarquía de memoria que se incluyen en el fichero *--mem-config* suministrado durante la simulación (para más información sobre la parametrización de Multi2Sim, véase el apartado 3.1.2). Como puede observarse, el tamaño y geometría de la cache de nivel 1 (L1) permanecerá inalterado a lo largo de todo el ciclo de pruebas, y esta configuración se usa tanto para el modelo de procesador de un solo núcleo (fig. 4.1) como en la versión multinúcleo

(figura 4.2). En el caso de la memoria cache de nivel 2 (L2), no se fijan ni la capacidad total ni el número de vías, siendo estos dos parámetros las principales variaciones efectuadas durante los experimentos. Por último, la latencia de memoria principal es de 300 ciclos, similar a la que se puede medir en los procesadores actuales de altas prestaciones. Los parámetros comunes de la jerarquía de cache se basan en *Analyzing the Optimal Ratio of SRAM Banks in Hybrid Caches* [5].

La segunda sección de la figura 4.3 corresponde a los valores de la arquitectura x86 que se indican en el fichero *--x86-config*. En general, se han establecido valores superiores a los predeterminados en Multi2Sim tanto para los tamaños de las colas como para el número de registros, a fin de evitar la aparición de cuellos de botella diferentes al subsistema de memoria que pudieran llevar a conclusiones erróneas.

Hay que reseñar que, a fin de homogeneizar las condiciones de toma de medidas a lo largo de las distintas simulaciones, se ha establecido un valor fijo de quinientos millones de instrucciones para *FastForward* en todos los experimentos. Dado que este valor se establece por cada núcleo de ejecución, en las simulaciones multinúcleo el valor marcado en *--x86-config* será de dos mil millones (4 x 500M).

La tercera sección corresponde a otros dos parámetros, indicados directamente en el comando de ejecución de Multi2Sim. Estos son el número máximo de instrucciones que cada núcleo ejecutará de su carga de trabajo, y la cantidad de datos que deseamos obtener en el informe resumen de salida de las simulaciones, que establecemos al nivel máximo (simulación detallada, *detailed*).

4.1 Estudio de caracterización de la carga

Para la consecución del principal objetivo del proyecto precisamos conocer la naturaleza y rendimiento individual de cada una de las aplicaciones en procesadores con un solo núcleo. Estos valores nos servirán como base para evaluar la variación experimentada al ejecutarlas en entornos multinúcleo.

Además, hemos de analizar cuidadosamente el comportamiento de las distintas aplicaciones a fin de seleccionar aquellas que se utilizarán como referencia en los posteriores experimentos con varios núcleos, debido principalmente a dos factores:

- En un procesador multinúcleo, los elementos de la jerarquía de memoria compartidos entre los distintos núcleos se convierten en cuellos de botella. En la jerarquía base con la que trabajamos, la memoria cache de nivel 2 (L2) pasa a ser compartida. Por tanto, al recibir de forma simultánea accesos provenientes de varios núcleos se producirán más reemplazos en L2, lo que provocará más fallos. Para poder analizar las pérdidas de prestaciones debidas a estos fallos, debemos seleccionar aquellas aplicaciones de trabajo (*benchmarks* de SPEC CPU2006) teniendo en cuenta sus necesidades en la cache de L2, ya que una elección inapropiada (por ejemplo, una aplicación con pocos accesos a la L2) apenas se vería afectada por la compartición de L2.

- El tiempo de ejecución en Multi2Sim se dispara en las simulaciones multinúcleo. Debido a la gran cantidad de simulaciones a realizar, es aconsejable que se realicen en un clúster con muchos nodos de procesamiento. Lamentablemente, el hecho de ser compartido y la política de prioridades, que penaliza las sobrecargas de trabajo (véase apartado 3.3.2), no resulta viable explotar todas las combinaciones posibles entre los distintos *benchmarks* de SPEC CPU2006.

Por tanto, en el primer experimento, para estudiar las necesidades de cada aplicación respecto a L2, procedemos a modelar una arquitectura de un solo núcleo con una jerarquía de cache conforme a la mostrada en la figura 4.1. Se ha seleccionado una memoria cache de nivel 2 de 1MB de capacidad, asociativa por conjuntos de 16 vías.

4.1.1 Modificaciones en Multi2Sim. Aciertos por vías de la cache

Para el estudio de caracterización de la carga, dispondremos disponer de información de salida sobre la distribución de los aciertos de los distintos *benchmarks* en las diferentes vías de la cache L2. Sin embargo, Multi2Sim no proporciona estos datos por defecto, por lo que hemos de acometer una primera modificación de su código fuente, que nos permita obtenerla. En este apartado se describen los cambios efectuados en el código del simulador.

Como los cambios que necesitamos sólo afectan a las estadísticas de salida de la jerarquía de memoria, los ficheros a modificar se encuentran en la carpeta *src/mem-system* (para más información, consúltese el apartado 3.1.1, relativo a la jerarquía de ficheros fuente de Multi2Sim).

Los cambios realizados, clasificados según los ficheros afectados, son:

cache.h. En este fichero de cabecera se declaran las diferentes clases de objeto que afectan a las memorias cache. En particular, la clase *cache_t* define el módulo de cache en sí mismo. Añadimos un nuevo atributo a la clase:

```
long long *way_hits;
```

Este atributo será un puntero a un array de tipo *long long*, que tendrá tantas posiciones como vías existan en ese módulo de cache, y almacenará en cada posición los aciertos que se hayan producido en esa vía. El tipo de datos C *long long* es el que utiliza Multi2Sim de forma genérica para valores enteros muy grandes (accesos, aciertos, fallos, número de ciclos, instrucciones, etc.).

cache.c. Este fichero contiene la implementación de la función *cache_create*, donde se construye el objeto *cache* de la clase *cache_t*. De esta forma, inicializamos aquí el atributo *way_hits* declarado en el fichero anterior, mediante la línea de código:

```
cache->way_hits = xcalloc(assoc, sizeof(long long));
```

La función *xcalloc* reserva memoria e inicializa (a ceros) tantas posiciones como necesitemos para almacenar *assoc* elementos que ocupen cada uno el tamaño de una variable de tipo *long long* (para lo cual usamos la llamada a *sizeof*). La variable *assoc* es un parámetro de entrada de la función *cache_create*, que indica el número de vías del módulo de cache. Tenemos así un array de tipo *long long* con tantas posiciones como vías tenga la memoria.

module.c. En la función *mod_find_block* de este fichero, se procesan los aciertos y fallos de cada módulo de cache, incrementando –entre otras operaciones– sendos contadores globales para las estadísticas de salida del informe de simulación de la jerarquía de memoria.

Cada módulo de cache cuenta con una lista enlazada de bloques ordenada según los contadores LRU del módulo. Por tanto, necesitamos declarar una variable entera –*count*–, y justo tras el contador general de hits, implementar un bucle que realice una

búsqueda en la lista enlazada hasta dar con el bloque en el que se produce el acierto, incrementando nuestro array en la posición que marque el contador cuando encontremos dicho bloque. El código final tiene la forma siguiente:

```
count = 0;
for(blk=cache->sets[set].way_head; blk; blk=blk->way_next)
{
        if (blk == &cache->sets[set].blocks[way])
        {
                cache->way_hits[count] += 1;
                break;
        }
        count++;
}
```

mem-system.c. Este fichero contiene el tratamiento central de los datos procedentes de todo el sistema de memoria. En particular, se realiza la impresión de todas las variables estadísticas al fichero de salida que en la simulación se ha marcado con el parámetro --mem-report, mediante llamadas a la función *fprintf*. Por tanto, incluimos un sencillo bloque de código para imprimir las posiciones de nuestro array:

```
if (cache)
{
        for (j=0; j < cache->assoc; j++)
                fprintf(f, "Hits, Way %d = %lld\n", j, cache->way_hits[j]);
}
```

La variable booleana (cache) viene proporcionada por el código estándar de Multi2Sim y permite comprobar si el módulo cuyas estadísticas estamos procesando corresponde a este tipo de memoria.

4.1.2 Cargas de trabajo

Como se explicó en la sección 3.2, se han utilizado los *benchmarks* de SPEC CPU2006 a modo de cargas de trabajo para los diferentes contextos de ejecución. Como se ha mencionado previamente, se han incluido binarios compilados para arquitectura x86 de los distintos *benchmarks*, ya que su uso aquí no pretende la evaluación del rendimiento que proporciona SPEC CPU, sino como cargas representativas de trabajo real.

En este experimento en particular, se han lanzado 29 trabajos e ejecución en el sistema de colas, tantos como *benchmarks* componen las SPEC CPU. Los trabajos eran idénticos entre sí salvo su fichero de contexto, que contenía en cada caso los parámetros de ejecución de un *benchmark* diferente. Todos compartían el mismo fichero de configuración de la jerarquía de cache, con un nivel 1 en arquitectura *Harvard*, de 64KB en total, y un nivel 2 unificado de 1MB, tal y como se describe en la introducción.

4.1.3 Resultados de la simulación

La figura 4.1.3.1 muestra la distribución porcentual de aciertos (*hits*) entre las diferentes vías de la cache L2, ordenadas por su posición en la cola LRU, siendo la primera vía la señalada con valor cero. Se han agrupado los resultados a partir de la quinta vía.

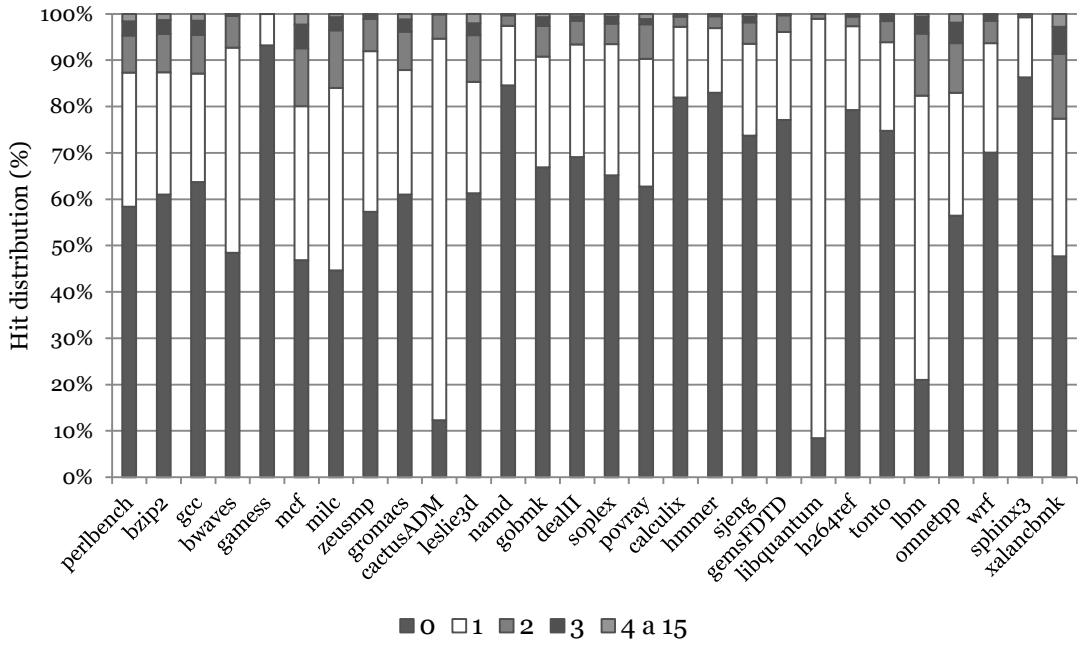

Figura 4.1.3.1**:** Hits por vía, cache L2 1MB de 16 vías. Vías por posición en la cola LRU.

Como puede observarse en la figura, la mayoría de los *benchmarks* concentran los aciertos en la primera vía de la cola LRU, con algunas excepciones (cactusADM, libquantum, lbm) que lo hacen en la segunda vía. Puede apreciarse una gran localidad, con una concentración en las dos primeras vías de la cola LRU del 80% y el 95% de los aciertos, según el *benchmark*. Puede apreciarse igualmente el desaprovechamiento generalizado de las vías superiores de la cache L2, toda vez que el porcentaje de aciertos en el conjunto de las vías 4 a 15 en la gráfica es residual.

A la vista de los resultados obtenidos podemos clasificar las aplicaciones en dos grandes grupos, aquellas que entre las dos primeras vías LRU concentran más del 90% de los aciertos (y que por lo tanto tendrán una *baja* penalización al compartir la memoria cache entre otras aplicaciones en un procesador multinúcleo) y el resto, que necesitan 3 o más vías para cubrir el 90% de aciertos (y que tendrán *a priori* una penalización *alta* en un entorno de cache compartida). Tendremos, así:

Penalización *baja* por vías de la cache: bwaves, gamess, zeusmp, cactusADM, namd, gobmk, dealII, soplex, povray, calculix, hmmer, sjeng, gemsFDTD, libquantum, h264ref, tonto, wrf, sphinx3.

Penalización *alta* por vías de la cache: perlbench, bzip2, gcc, mcf, milc, gromacs, leslie3d, lbm, omnetpp, xalancbmk.

La figura 4.1.3.2 muestra el índice MPKI (*Misses Per Kilo-Instruction*) ó fallos por cada mil instrucciones de la cache de datos de nivel 1. Recordemos que el MPKI de un módulo de memoria se obtiene mediante la fórmula:

$$MPKI = \frac{Misses \times 1000}{Committed\ Instructions}$$

Donde *Misses* denota el número total de fallos en el módulo de cache, y *Committed Instructions* es el número de instrucciones que han pasado por la fase de *Commit* en el ciclo de instrucción.

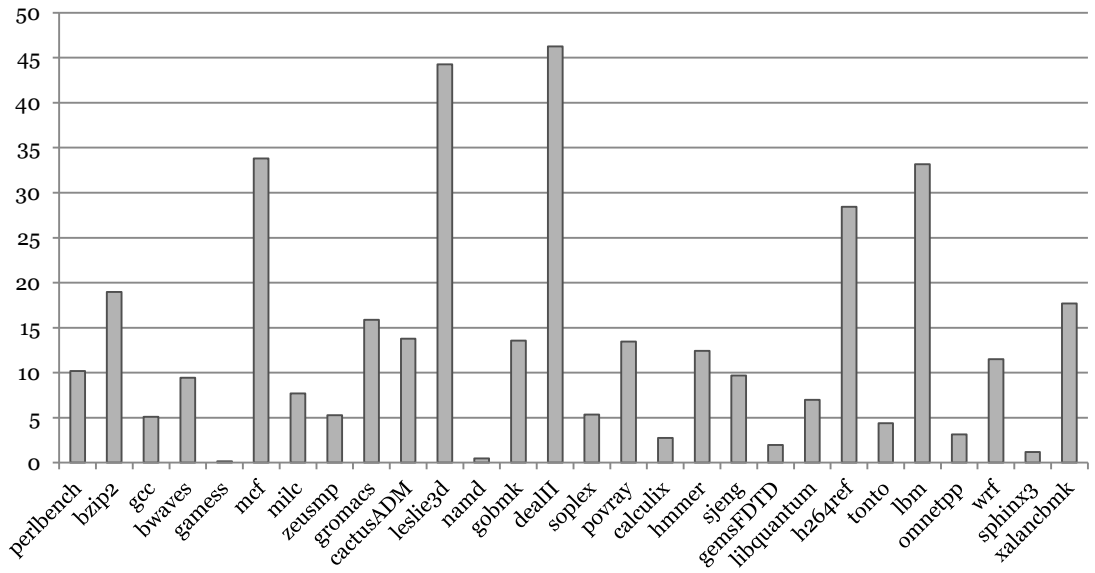

Figura 4.1.3.2: MPKI, cache L1 de datos.

Asumiendo que el ratio MPKI puede considerarse que es *bajo* cuando no alcanza un valor de 5, *medio* cuando su valor se sitúa entre 5 y 20, y *alto* cuando es mayor que 20, los resultados mostrados en la figura 4.1.3.2 nos permiten clasificar también las aplicaciones, esta vez en función del MPKI de la cache de datos de nivel 1.

Así, tendremos:

- **MPKI L1 bajo**: gamess, namd, calculix, gemsFDTD, tonto, omnetpp, sphinx3.
- **MPKI L1 medio**: perlbench, bzip2, gcc, bwaves, milc, zeusmp, gromacs, cactusADM, gobmk, soplex, povray, hmmer, sjeng, libquantum, wrf, xalancbmk.
- **MPKI L1 alto**: mcf, leslie3d, dealII, h264ref, lbm.

De forma análoga, la figura 4.1.3.3 muestra el valor del índice MPKI para los distintos *benchmarks* en la cache L2:

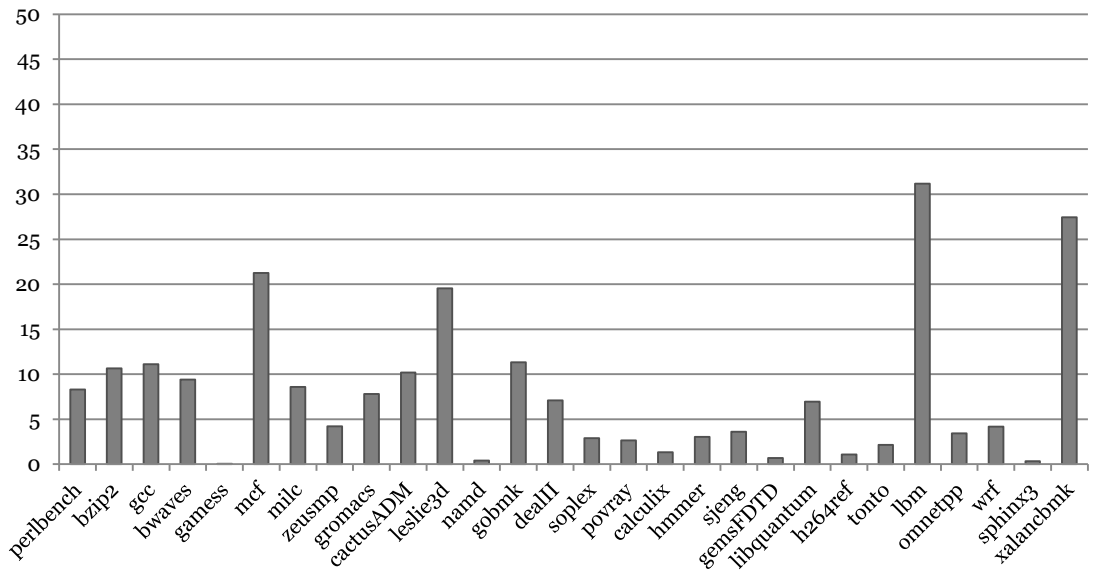

Figura 4.1.3.3: MPKI para la cache L2.

La comparativa de las gráficas señaladas en las figuras 4.1.3.2 y 4.1.3.3 permite observar las diferencias existentes en función del *benchmark* ejecutado. Núcleos como dealII muestran una acuciada diferencia entre el MPKI de la cache de nivel 1 y la de nivel 2. Otros, como lbm, presentan valores muy similares. Para el análisis de estos datos ha de tenerse en cuenta que el MPKI de la cache de nivel 2 incluye también los fallos de cache de instrucciones, toda vez que se trata de un módulo unificado para instrucciones y datos, mientras que el nivel 1 de la jerarquía de cache presenta una arquitectura *Harvard*, con módulos separados para cada conjunto.

Considerando de nuevo que el MPKI es *bajo* cuando no alcanza un valor de 5, *medio* cuando su valor se sitúa entre 5 y 20, y *alto* cuando es mayor que 20, los resultados mostrados en la figura 4.1.3.3 nos permiten realizar una tercera clasificación de las aplicaciones, en función del MPKI de la cache de datos de nivel 2, que será:

- **MPKI L2 bajo**: gamess, zeusmp, namd, soplex, povray, calculix, hmmer, sjeng, gemsFDTD, h264ref, tonto, omnetpp, wrf, sphinx3.

- **MPKI L2 medio**: perlbench, bzip2, gcc, bwaves, milc, gromacs, cactusADM, leslie3d, gobmk, dealII, libquantum.

- **MPKI L2 alto**: mcf, lbm, xalancbmk.

La figura 4.1.3.4 muestra la tasa de acierto o *Hit Ratio* de la memoria cache de nivel 2. La tasa de acierto de un módulo de memoria se obtiene mediante la fórmula:

$$HitRatio = \frac{Aciertos}{Accesos\ totales}$$

Al tratarse de un dato promedio, los valores de la tasa de acierto oscilan entre 0 y 1, suponiendo este último valor una cache que contuviese toda la información que requiriese el procesador.

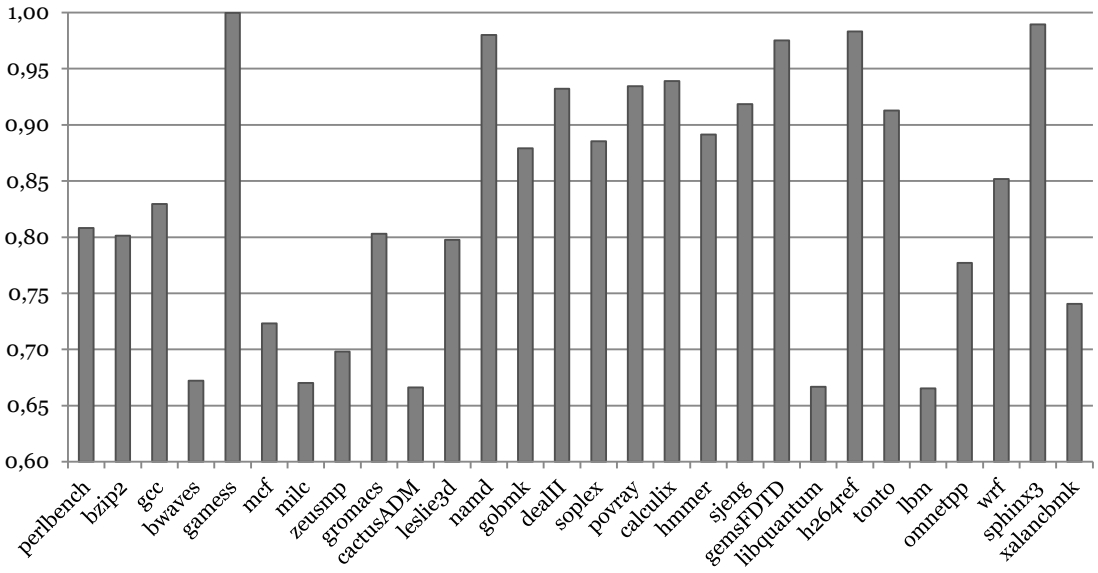

Figura 4.1.3.3: Tasa de acierto o HitRatio, cache L2.

Por último, calcularemos el IPC (*Instructions Per Cycle*) ó instrucciones por cada ciclo de los diferentes *benchmarks* de SPEC CPU2006. El IPC se obtiene de la fórmula:

$$IPC = \frac{Instrucciones}{Ciclos}$$

Sin embargo, para nuestro estudio nos basaremos en el cálculo del IPC tomando como base el número de microinstrucciones (internas de la implementación de la arquitectura x86 del simulador) que han pasado por la fase de *commit*. El nuevo índice IPC se obtendrá de forma análoga, con la fórmula:

$$IPC = \frac{\mu Instrucciones}{Ciclos}$$

La figura 4.1.3.4 muestra el IPC obtenido en cada carga del experimento final con un solo núcleo, tomando como base de cálculo el número de microinstrucciones ejecutadas.

Como puede observarse en la figura, el IPC varía entre valores inferiores a 0.5 microinstrucciones por ciclo para *benchmarks* como lbm, xalancbmk o mcf, hasta valores por encima de dos microinstrucciones por ciclo observados en gamess y gemsFDTD.

Si consideramos un IPC alto aquel cuyo valor está por encima de 1 microinstrucciones por ciclo, un IPC medio si se encuentra comprendido entre 0.5 y 1, y un IPC bajo si es inferior a 0.5, podemos realizar también una clasificación de las aplicaciones, obteniendo:

- **IPC bajo**: bzip2, mcf, cactusADM, leslie3d, libquantum, lbm, xalancbmk.

- **IPC medio**: perlbench, gcc, bwaves, milc, zeusmp, gromacs, gobmk, dealII, sjeng, wrf.

- **IPC alto**: gamess, namd, soplex, povray, calculix, hmmer, gemsFDTD, h264ref, tonto, omnetpp, sphinx3.

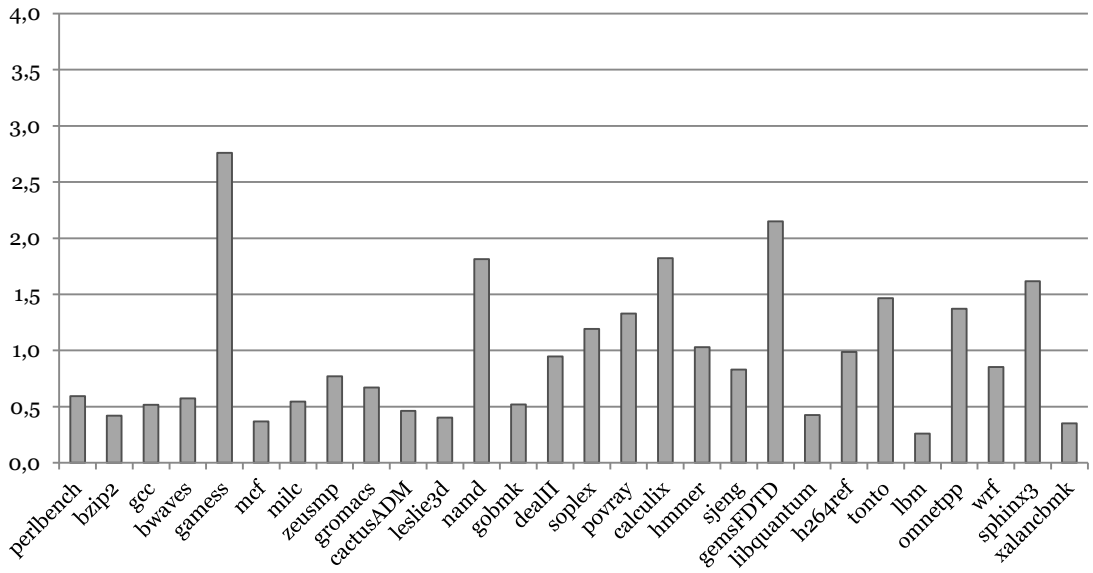

Figura 4.1.3.4: IPC por aplicación.

Los resultados obtenidos en esta simulación nos ofrecen un punto de partida sólido para la planificación de los siguientes experimentos, toda vez que posibilitan el clasificar los *benchmarks* según su necesidad en vías de la cache L2, lo que a su vez nos permitirá aventurar su comportamiento cuando hayan de compartir esta cache L2 con otras aplicaciones en simulaciones multinúcleo.

Los siguientes experimentos tendrán como base estos resultados y las clasificaciones realizadas a partir de ellos, para conformar las cargas de trabajo de las simulaciones con procesadores multinúcleo.

4.2 Impacto de la geometría de la cache de nivel 2

Tras el estudio de caracterización de la carga efectuado en el apartado 4.1, el siguiente paso es realizar sucesivas simulaciones variando la geometría de la memoria cache de nivel 2 en el procesador de un solo núcleo para ver cómo las variaciones estudiadas afectan a cada aplicación.

Por tanto, seleccionamos distintas geometrías que sustituirán a *l2topo* en el fichero de configuración de memoria de Multi2Sim (véase apartado 3.1.2). Nótese que será preciso realizar una simulación distinta por cada aplicación para cada una de las geometrías. Cabe recordar que Multi2Sim permite automatizar el lanzamiento de múltiples simulaciones simultáneas mediante *scripts* de Shell, tal y como se indicó en el anteriormente. Las geometrías simuladas son las siguientes:

- 512KB y 16 vías
- 512KB y 8 vías
- 256KB y 16 vías
- 256KB y 8 vías
- 128KB y 16 vías
- 128KB y 8 vías

Tal y como vimos al describir el proceso de simulación de sistemas informáticos con Multi2Sim (apartado 3.1.2), contamos con tres parámetros relativos al tamaño en el fichero de configuración: tamaño del bloque (*BlockSize*), número de conjuntos (*Sets*) y número de vías (*Assoc*). Dado que el primero de ellos permanecerá fijo a 64 Bytes, variaremos el número de conjuntos para obtener las distintas capacidades de cache. Así, la cache L2 de 512 KB y 16 vías tendrá un valor de *Sets* definido de la forma:

$$Conjuntos = \frac{512KB}{64B \times 16vías} = 512$$

El objetivo de este experimento es medir la pérdida de prestaciones experimentada por las distintas aplicaciones ante tamaños cada vez menores de la memoria cache de nivel 2, en comparación con los datos obtenidos en el experimento anterior con una cache L2 de 1MB de capacidad y 16 vías por conjunto.

4.2.1 Resultados de la simulación

La figura 4.2.1 muestra la distribución de aciertos (*hits*) entre las vías de la cola LRU en la memoria cache de nivel 2, para las geometrías de 512KB y 16 vías (superior), 256KB y 16 vías (centro) y 128KB y 16 vías (inferior).

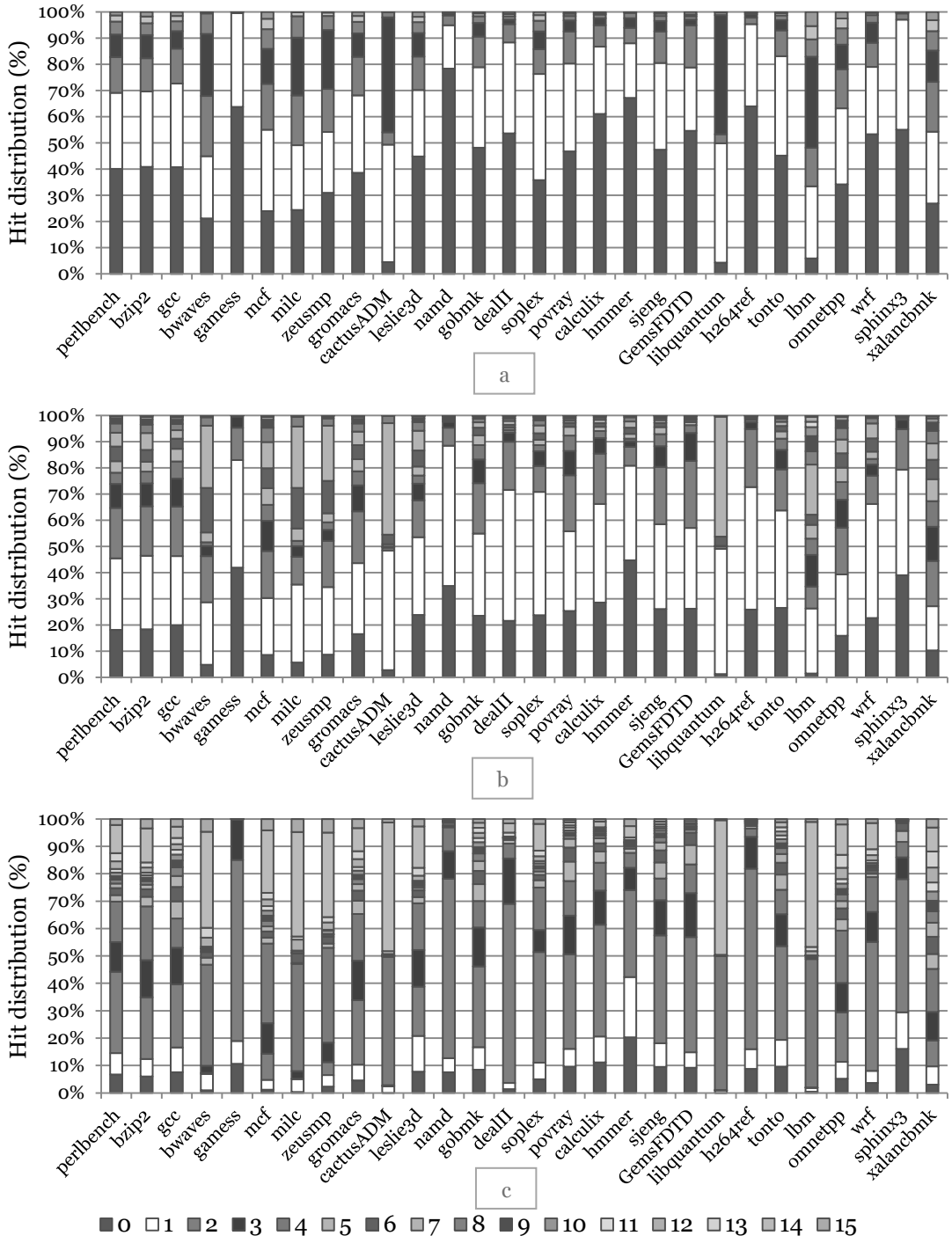

Figura 4.2.1: Hits por vía, L2 512KB 16 vías (a), L2 256KB 16 vías (b), L2 128KB 16 vías (c). Vías por posición en la cola LRU.

La figura 4.2.1 muestra un aumento progresivo en la distribución de aciertos entre las vías, conforme disminuye el tamaño de la cache, manteniendo fijo a 16 el número de vías por conjunto de la cache. Así, en la gráfica superior de la figura, correspondiente a la memoria cache de 512KB y 16 vías, las cuatro primeras vías concentran más del 90% de los aciertos en la totalidad de las cargas, mientras que en la gráfica central (cache de 256KB) y especialmente en la inferior (cache de 128KB), los aciertos se encuentran más repartidos entre las vías de la cola LRU.

Comparando estos resultados con los obtenidos durante el estudio de caracterización (1MB y 16 vías), podemos analizar cómo afecta la reducción del tamaño de la memoria cache a la distribución de aciertos en las aplicaciones. Así, se observan varias tendencias de comportamiento entre los distintos *benchmarks*.

De entre aquellos con una penalización *baja* por vías de la cache (aplicaciones que necesitan dos vías o menos para cubrir el 90% de aciertos), las aplicaciones que con la cache de 1MB tenían un porcentaje de aciertos superior al 70% ya en la primera vía de la cola LRU (gamess, namd, calculix, hmmer, sjeng, gemsFDTD, h264ref, tonto, wrf, xphinx3) ven reducidas, de forma general, los aciertos en esa primera vía entre un 10 a un 20% del total cuando se usa la cache de 512KB, aunque cuatro de ellas (gamess, namd, h264ref y sphinx3) siguen sumando alrededor de un 90% de aciertos totales entre las dos primeras vías. Con la cache de 256KB se sigue reduciendo el porcentaje de aciertos en las dos primeras vías para estas diez aplicaciones, aunque ambas vías continúan concentrando más del 50% de los aciertos. Esta tendencia se rompe con la cache de 128KB, cuyos resultados muestran, además de una mayor distribución de los aciertos entre las vías, que el papel preponderante pasa a vías posteriores de la cola LRU (concretamente, de la segunda a la quinta vía).

Por otro lado, analizando los *benchmarks* que necesitando dos vías o menos para cubrir el 90% de aciertos con la cache de 1MB concentraban estos en la segunda vía de la cola LRU (cactusADM, libquantum), se observa que con la cache de 512KB este 90% se concentra entre la segunda y cuarta vías, siendo la cuarta vía sustituida por la quinta en las geometrías de 256KB y 128KB.

Por su parte, los *benchmarks* con una penalización *alta* por vías de cache (los que en el estudio de caracterización precisaban más de dos vías para cubrir el 90% de aciertos), también muestran dos comportamientos distintos en función de si concentraban los aciertos en la primera o segunda vía de la cola LRU. Los primeros (perlbench, bzip2, gcc, mcf, milc, leslie3d, omnetpp, xalancbmk) ven reducida con la cache de 512KB el porcentaje de aciertos en la primera vía a valores cercanos al 40%, absorbiendo la segunda vía solo de forma parcial los aciertos perdidos por la primera, incrementándose así de forma sustancial los aciertos en la tercera vía de la cola LRU. Con la cache de 256KB, la importancia de la tercera vía se incrementa, hasta el punto de repartirse los aciertos de forma aproximadamente igual entre las vías 0,1 y 2 de la figura 4.2.1.b. Con la cache de 128KB, la tercera vía de la cola LRU pasa a concentrar un mayor número de aciertos que cualquier otra vía, si bien su proporción de aciertos es netamente inferior a la que concentraba inicialmente la primera vía.

El segundo grupo (formado solamente por lbm), que inicialmente concentraba los aciertos en la segunda vía de la cola LRU, reparte estos entre la segunda y cuarta vías en la cache de 512KB y entre la segunda y octava en la cache de 256KB –obsérvese la gran

distribución de aciertos entre las vías intermedias–, y entre la tercera y sexta vías en la cache de 128KB, con una concentración notable, dado que suman más del 90% de aciertos.

La figura 4.2.2 muestra la distribución de aciertos (*hits*) entre las vías de la cola LRU en la memoria cache de nivel 2, para las geometrías de 512KB y 8 vías (superior), 256KB y 8 vías (centro) y 128KB y 8 vías (inferior).

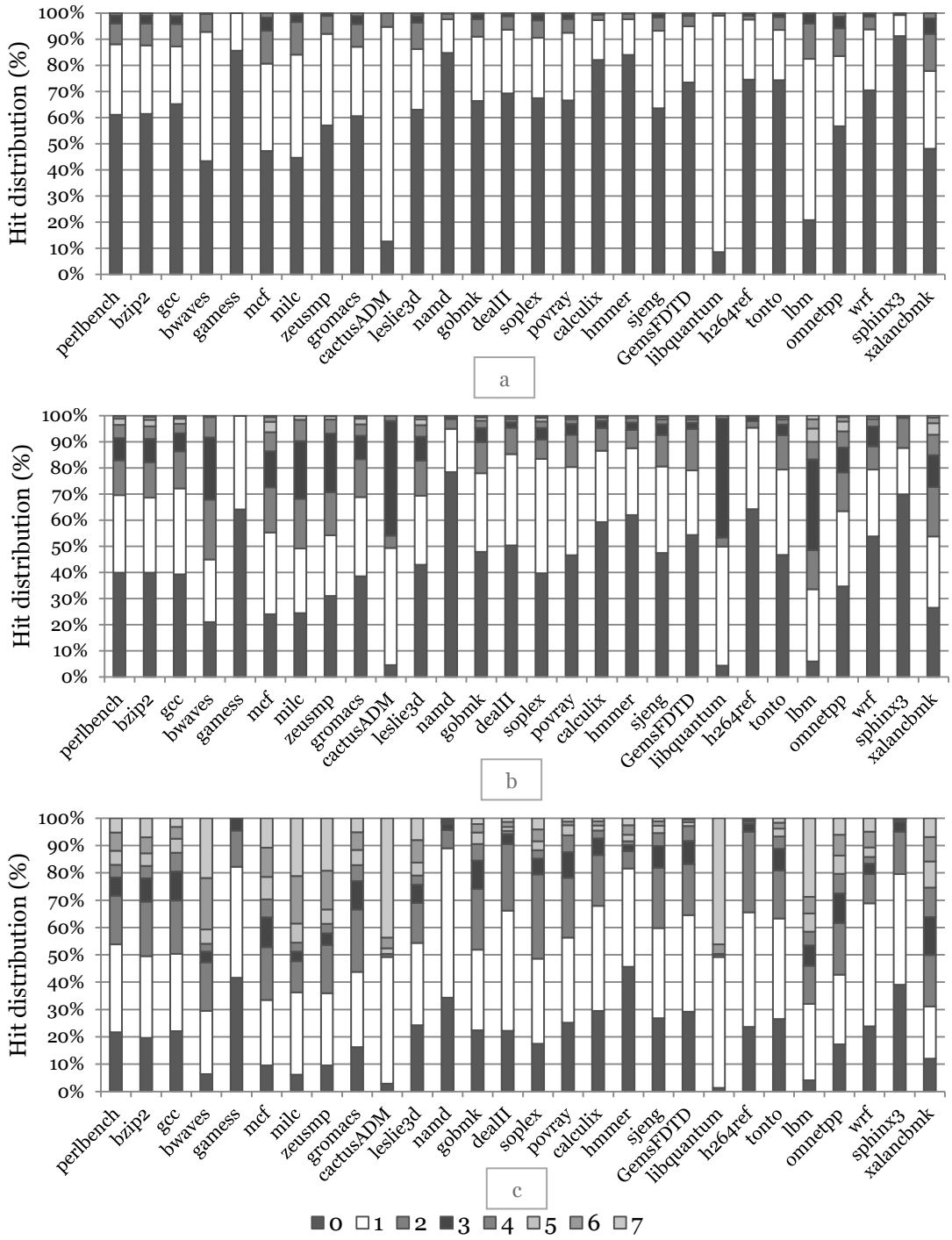

Figura 4.2.2: Hits por vía, L2 512KB 8 vías (a), L2 256KB 8 vías (b), L2 128KB 8 vías (c). Vías por posición en la cola LRU.

Comparando la figura 4.2.2 con la 4.2.1, podemos analizar cómo afecta la reducción de la asociatividad a la distribución de aciertos en las vías de la cache L2. Así, al comparar la gráfica *a* en la figura 4.2.1 (cache de 512KB y 16 vías) con la gráfica *a* en la figura 4.2.2 (cache de 512KB y 8 vías), observamos un reagrupamiento de los aciertos en las primeras vías de la cache en esta segunda gráfica, consecuencia de la disminución del número de vías disponibles en cada conjunto.

La comparativa de los resultados de las cache de 256KB entre ambas figuras muestra también una mayor concentración de aciertos en las tres primeras vías de la cola LRU en la figura 4.2.2.b (cache de 256KB y 8 vías). En concreto, aplicaciones como perlbench, bzip2 o gromacs, que suman una concentración superior al 80% entre las tres vías en la cache de 256KB y 8 vías, obtienen una tasa total de aciertos para las mismas vías en la cache de 256KB y 16 vías de alrededor del 65%. También puede observarse la misma distribución de aciertos en las vías segunda y cuarta (señaladas por 1 y 3, respectivamente, en las gráficas), para los *benchmarks* cactusADM o libquantum. Nótese que con la cache de 1MB y 16 vías estas aplicaciones concentraban sus aciertos en la segunda vía de la cola LRU.

En cuanto a las gráficas correspondientes a las geometrías de 128KB (figuras 4.2.1.c y 4.2.2.c), se observa un comportamiento similar al observado para las otras geometrías, consistente en una mayor concentración de aciertos en las primeras vías de la cola LRU, para la cache de 8 vías por conjunto con respecto a la cache de igual tamaño y doble número de vías por conjunto.

La figura 4.2.7 muestra las gráficas de los valores MPKI de la cache L1 de datos, para la geometría de cache L2 de 512KB y 16 vías. Nótese que los valores de cada *benchmark* para las distintas geometrías de L2 son a grandes rasgos similares entre ellos, ya que la geometría de la cache L1 no se ha visto modificada en el experimento, por lo que se ha optado por omitir las gráficas correspondientes al resto de geometrías.

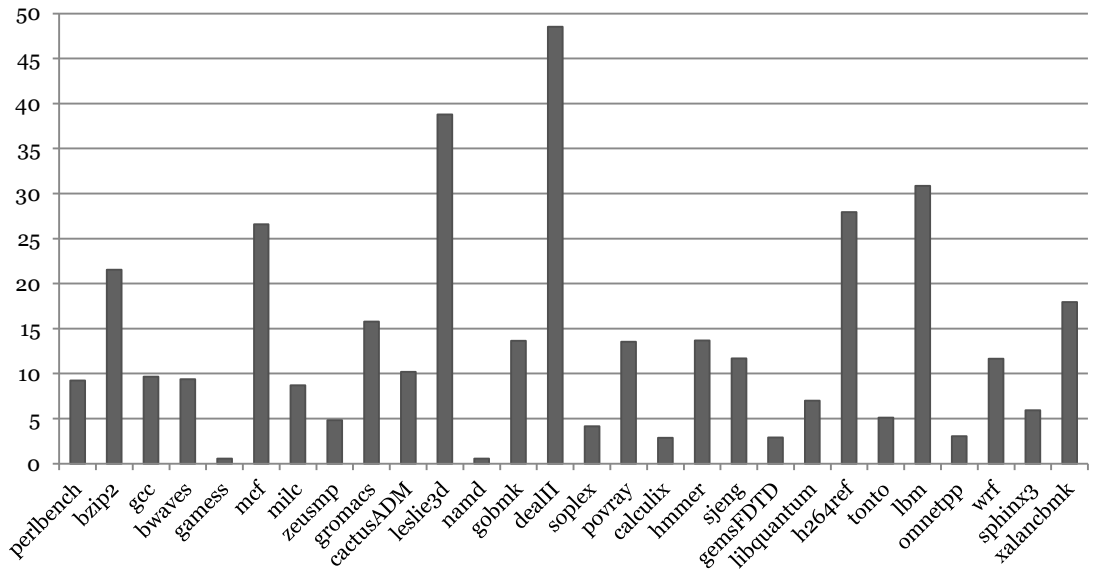

Figura 4.2.7: MPKI cache de nivel 1, cache L2.

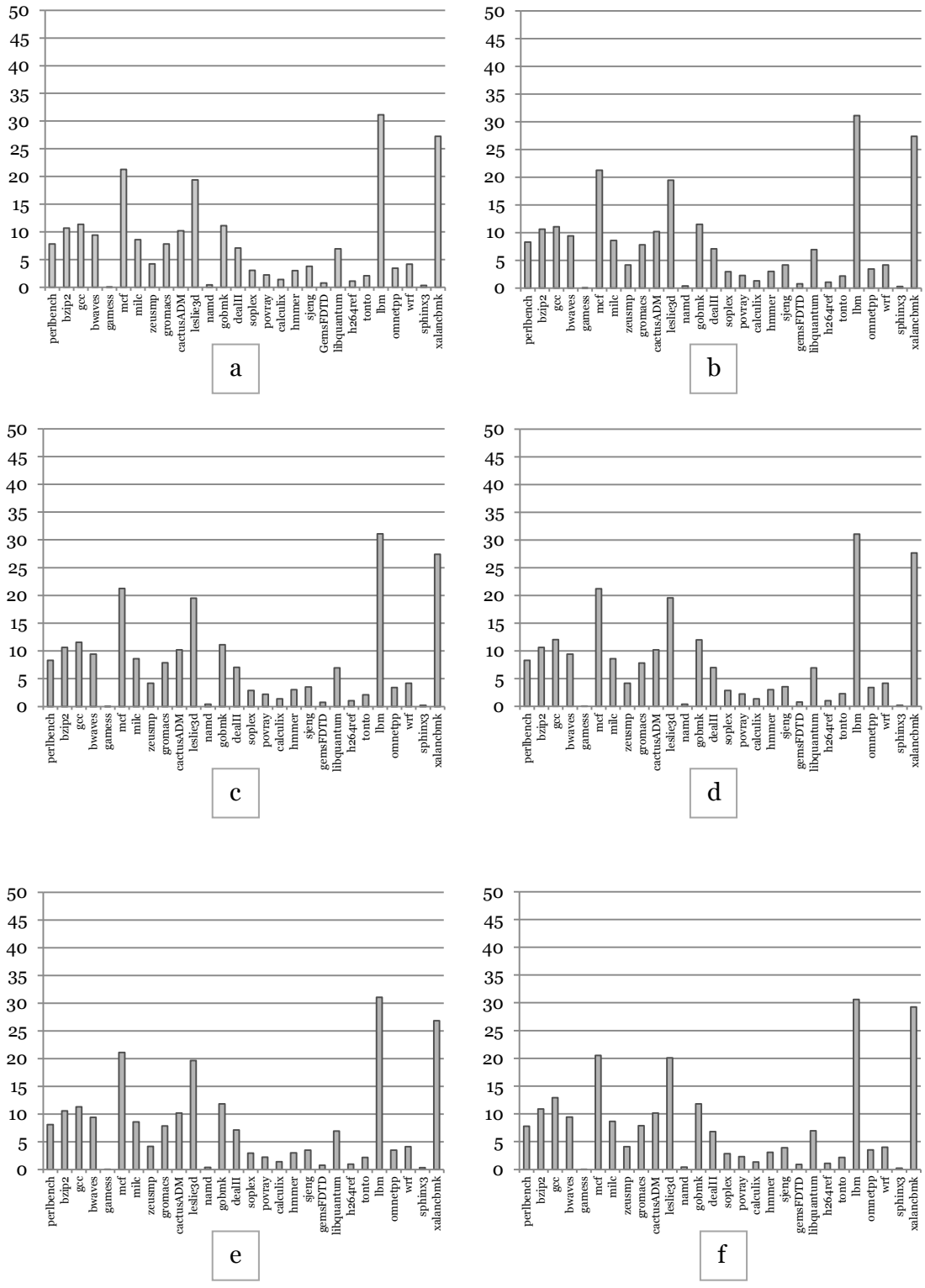

Figura 4.2.8: MPKI cache de nivel 2, cache L2 512KB y 16 vías (a), cache L2 512KB y 8 vías (b), cache L2 256KB y 16 vías (c), cache L2 256KB y 8 vías (d), cache L2 128KB y 16 vías (e), cache L2 128KB y 8 vías (f).

La figura 4.2.8 muestra los valores de MPKI de la cache de nivel 2, para cada una de las seis geometrías analizadas. A la vista de las gráficas de la figura, no es posible afirmar que exista un incremento del número de fallos por cada mil instrucciones conforme disminuimos el tamaño de la memoria cache. Sin embargo, el análisis de los datos numéricos obtenidos permite inferir pequeñas variaciones en este sentido. Así, bzip2 incrementa su MPKI desde un valor de 10.66 (L2 de 512KB y 16 vías) hasta 10.87 (L2 de 128KB y 8 vías), xalancbmk pasa de un MPKI de 27.25 a 28.38 y gromacs pasa de 7.81 a 7.88, para las mismas geometrías de la cache de nivel 2.

De forma análoga, a igualdad de tamaño total de la cache L2, se observa un incremento del MPKI al disminuir el número de vías de la cache. Así, leslie3d pasa de un valor de 19.37 (L2 512KB, 16 vías) a 19.42 (L2 512KB, 8 vías) y povray pasa de un MPKI de 2.23 (L2 512KB, 16 vías) a 2.28 (L2 512KB, 8 vías).

A tenor de los resultados obtenidos, puede afirmarse que existe una relación inversa entre el tamaño y asociatividad de la cache de nivel 2, y el ratio MPKI. A menor tamaño de la memoria cache, mayor el número de fallos de acceso por cada mil instrucciones, y a igualdad de tamaño de la cache, un menor número de vías provoca un mayor número de fallos, consecuencia ambos incrementos de la mayor cantidad de fallos por reemplazos en la cache. Dado que el índice MPKI guarda a su vez una relación inversa con las prestaciones de la memoria (a mayor número de fallos, mayor penalización debido al acceso a niveles superiores y más lentos de la jerarquía de memoria), se establece así una relación inversa entre el tamaño y asociatividad de la cache y las prestaciones generales del computador.

Para terminar el análisis con distintas geometrías de cache L2 en simulación con un solo núcleo, obtendremos el IPC (*Instructions Per Cycle*) tomando de nuevo, al igual que en el primer experimento (véase apartado 4.1.3), las microinstrucciones que han realizado la fase de *commit* como base del cálculo.

La figura 4.2.9 muestra los valores del IPC de las distintas cargas de trabajo, para cada una de las seis geometrías analizadas. La gráfica *a* muestra el IPC de la memoria cache L2 de 512KB y 16 vías. Los valores son similares a los obtenidos con la cache L2 de 1MB y 16 vías (figura 4.1.3.4), ya que los datos muestran un descenso en el IPC del orden de milésimas con respecto a los obtenidos en el primer experimento.

La gráfica *b* muestra los resultados del IPC para la memoria cache L2 de 512KB y 8 vías. De nuevo, no se aprecian diferencias significativass, ya que los datos muestran un descenso muy leve en el IPC (como ejemplo, gromacs obtiene un IPC de 0.3284 para la cache L2 de 512KB y 16 vías, y un IPC de 0.3280 para la cache de 512KB y 8 vías).

La gráfica *c* muestra los resultados del IPC para la memoria cache L2 de 256KB y 8 vías. La pérdida de IPC resulta ya apreciable en algunos *benchmarks*. Obsérvese el descenso de gamess, que pasa de un valor de 2.8417 para la cache de 512KB y 16 vías, a 2.7530 para la cache de 256KB y 8 vías.

La gráfica *d* muestra los resultados del IPC para la cache L2 de 256KB y 8 vías. El descenso del IPC con respecto a los resultados con las caches de mayor tamaño resulta visible en determinados *benchmarks*, como gamess y tonto.

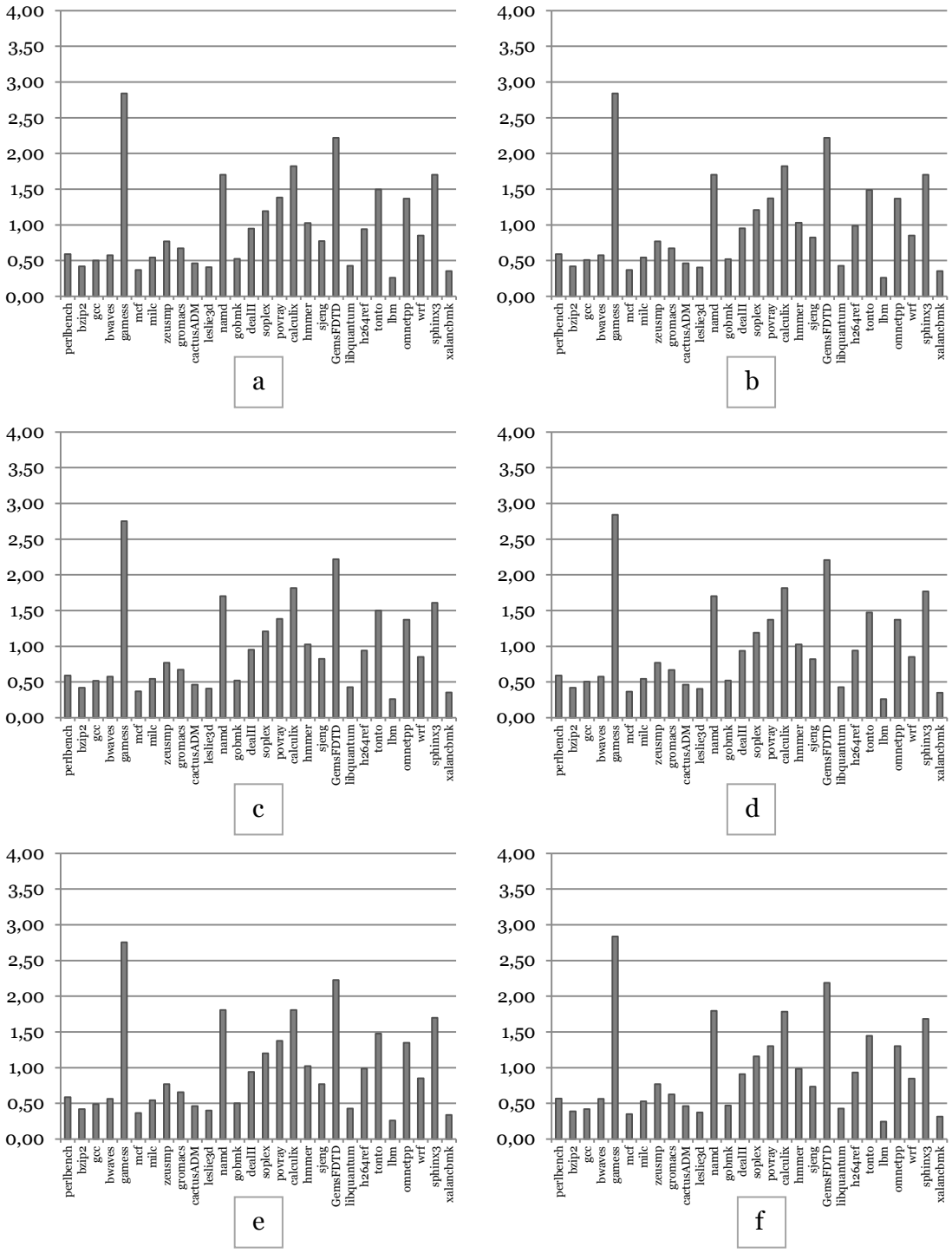

Figura 4.2.9: IPC por microinstrucciones, cache L2 512KB y 16 vías (a), cache L2 512KB y 8 vías (b), cache L2 256KB y 16 vías (c), cache L2 256KB y 8 vías (d), cache L2 128KB y 16 vías (e), cache L2 128KB y 8 vías (f).

La gráfica *e* muestra los resultados del IPC para la memoria cache de 128KB y 16 vías. Comparando los resultados obtenidos con la cache de 1MB y 16 vías (figura 4.1.3.4, apartado 4.1.3), se observa de forma más clara la pérdida en IPC que para las memorias cache de 512KB y 256KB. Así, cactusADM pasa de un IPC de 0.4618 (L2 de 1MB) a 0.2229 (L2 de 128KB), con el mismo número de vías pero una cache de 128KB supone

1/8 del tamaño de la cache de 1MB. De forma análoga, la comparativa de los datos obtenidos con esta geometría y las caches de 512KB y 256KB arrojan resultados inferiores para la cache de 128KB y 16 vías.

Por último, la gráfica *f* muestra los resultados del IPC para la memoria cache L2 de 128KB y 8 vías. Aunque no puede apreciarse fácilmente en la gráfica, los valores del IPC son levemente inferiores a los obtenidos por la cache L2 de 128KB y 16 vías (gráfica *e*), conformando así los valores más bajos de las seis geometrías de cache. Así, wrf pasa de un IPC de 0.8502 para la cache de 128KB y 16 vías, a un IPC de 0.8476 para la cache de 128KB y 8 vías, mientras que este mismo *benchmark* obtiene un IPC de 0.8524 para la cache L2 de 1MB y 16 vías, y de 0.8514 para la cache de 512KB y 8 vías.

A la vista de los resultados obtenidos en el cálculo del IPC para las distintas geometrías, se infiere una relación directa entre el tamaño de la memoria cache y el índice IPC, así como entre este ratio y el número de vías de la cache. Dado que el IPC, a diferencia del MPKI, guarda una relación directa con las prestaciones del computador (a mayor número de instrucciones ejecutadas por cada ciclo de reloj, mayor rendimiento del procesador), puede aseverarse, de forma análoga a lo afirmado para el análisis del MPKI, que el tamaño y asociatividad de la cache guardan una relación directa con las prestaciones del procesador y, por ende, del computador.

4.3 Evaluación de las prestaciones en procesadores multinúcleo

Tras el análisis del comportamiento y prestaciones tanto de la cache de nivel 2 como de las aplicaciones en un procesador con un solo núcleo, se traslada el estudio al ámbito de un procesador multinúcleo, similar a los existentes en el mercado actual. Tal y como se señaló al comienzo del capítulo, el procesador a modelar contará con cuatro núcleos, cada uno de ellos con un nivel 1 de cache privado, en arquitectura *Harvard*, siendo la cache de nivel 2 unificada común a todos los núcleos (véase las figuras 4.2 y 4.3 para una descripción más detallada de las características del procesador multinúcleo modelado).

Indudablemente, el comportamiento del nivel 1 (L1) de la jerarquía de cache en este computador multinúcleo será idéntico al de la simulación con un solo núcleo, toda vez que cada núcleo dispone de sus propios módulos de memoria cache L1 en exclusiva. Por tanto, el objetivo a analizar será forzosamente la cache compartida de nivel 2 (L2), que actuará aquí como elemento *cuello de botella*.

Al igual que en el experimento titulado *Impacto de la geometría de la cache de nivel 2*, se realizarán diferentes simulaciones, variando el tamaño y número de vías por conjunto de la cache L2. En particular, se modelarán las mismas geometrías que en dicho experimento:

- 512KB, 16 vías
- 512KB, 8 vías
- 256KB, 16 vías
- 256KB, 8 vías
- 128KB, 16 vías
- 128KB, 8 vías

Al ser el objetivo de este estudio el evaluar la jerarquía de cache, nos centraremos no en las prestaciones intrínsecas del procesador multinúcleo, cuyo rendimiento general será superior al de un solo núcleo, sino en la variación de las prestaciones experimentadas por las aplicaciones ejecutadas en cada uno de los núcleos que componen el procesador.

4.3.1 Selección de las cargas de trabajo

Para la correcta interpretación de los resultados de las simulaciones en un procesador multinúcleo, es crucial una correcta selección de las cargas de trabajo que se ejecutarán, toda vez que el experimentar con cargas con comportamiento y prestaciones muy dispares podría llevar a conclusiones erróneas.

En el estudio de caracterización de la carga, tomando como base una memoria cache L2 de 1MB y 16 vías, se realizó una clasificación de los *benchmarks* de SPEC CPU2006 en

base a la distribución de aciertos en las vías de la cache L2, así como a sus valores de MPKI e IPC (véase el apartado 4.1).

La clasificación por MPKI y distribución de aciertos por vías de la cache nos permiten identificar el comportamiento de cada carga en lo que a acceso a recursos del sistema de memoria se refiere, mientras que la clasificación por IPC nos permite identificar qué aplicaciones tienen –a igualdad de condiciones– un rendimiento mayor y, por tanto, evaluar de forma más rápida la pérdida de prestaciones sufridas en un entorno compartido. Por lo tanto, tomando como base las categorías establecidas por MPKI y necesidad de vías de la cache L2, conformamos ahora cinco mezclas de carga diferentes, con los criterios que se señalan:

1. **gromacs, gromacs, bzip2, bzip2**

   Esta mezcla utiliza dos cargas diferentes: gromacs y bzip2. Ambas de penalización *media* en cuanto al ratio MPKI tanto de nivel 1 como de nivel 2, y de penalización *alta* en el uso de las vías de la cache. Nótese que dos núcleos ejecutarán por separado dos aplicaciones diferentes de gromacs y los otros dos, sendas aplicaciones de bzip2. Al mezclarse cuatro cargas con categoría *media* en MPKI y *alta* en vías, debe experimentarse una considerable penalización del rendimiento individual de cada carga al ejecutarse en el procesador multinúcleo.

2. **dealII, hmmer, gromacs, gromacs**

   Esta mezcla utiliza tres *benchmarks* diferentes: dealII (de penalización *alta* en el MPKI de nivel 1 y *media* en nivel 2, y penalización *baja* en cuanto a vías de la cache), hmmer (MPKI L1 *media*, MPKI L2 *baja* y penalización por vías *baja*) y gromacs (MPKI L1 *media*, MPKI L2 *media* y penalización por vías de la cache *baja*). Se repite el uso de gromacs en los dos últimos núcleos para una mejor comparativa con los resultados obtenidos en la primera mezcla de cargas. Por su naturaleza, la penalización en el rendimiento individual de dealII y hmmer debería ser inferior a la experimentada por gromacs, si bien todas las cargas habrán de perder prestaciones con respecto al procesador de un solo núcleo.

3. **h264ref, leslie3d, mcf, mcf**

   Esta mezcla utiliza también tres *benchmarks* diferentes: h264ref (de penalización *alta* en el MPKI de nivel 1 y *baja* en nivel 2, y penalización *baja* en cuanto a vías de la cache), leslie3d (MPKI L1 *alta*, MPKI L2 *media* y penalización por vías *alta*) y mcf (MPKI L1 *alta*, MPKI L2 *alta* y penalización por vías de la cache *alta*). La pérdida de prestaciones de h264ref será *a priori* inferior al de las otras dos cargas, especialmente mcf, que en principio debería verse mucho más perjudicada, al tener penalización alta en las tres categorías. Por este motivo, se ejecutará mcf por separado en dos de los núcleos.

4. **povray, sjeng, sjeng, wrf**

   Esta mezcla utiliza también tres *benchmarks* diferentes: povray (de penalización *media* por MPKI de nivel 1 y *baja* en nivel 2, y penalización *baja* en cuanto a vías de la cache), sjeng (MPKI L1 *media*, MPKI L2 *baja* y penalización por vías *baja*) y wrf (MPKI L1 *media*, MPKI L2 *baja* y penalización por vías de la cache *baja*). Esta mezcla es bastante homogénea en cuanto a su naturaleza, y está formada por *benchmarks* que experimentarán *a priori* una pérdida de prestaciones relativamente baja. En esta ocasión se repite carga de trabajo en los núcleos segundo y tercero (sjeng).

5. **dealII, hmmer, namd, sphinx3**

   Esta es la única mezcla formada por cuatro *benchmarks* diferentes: dealII (de penalización *alta* en el MPKI de nivel 1 y *media* en nivel 2, y penalización *baja* en cuanto a vías de la cache), hmmer (MPKI L1 *media*, MPKI L2 *baja* y penalización por vías *baja*), namd (MPKI L1 *baja*, MPKI L2 *baja* y penalización por vías de la cache *baja*) y sphinx3 (MPKI L1 *baja*, MPKI L2 *baja* y penalización por vías de la cache *baja*). Tiene en común las dos primeras cargas de la mezcla número 2.

La ejecución de la misma aplicación en varios núcleos distintos, como se ha hecho en las cuatro primeras mezclas, permitirá además identificar posibles anomalías producidas durante el modelado o ejecución de las simulaciones.

4.3.2 Modificaciones en Multi2Sim. IPC por aplicación

Para analizar correctamente las prestaciones de las aplicaciones en el computador multinúcleo y obtener resultados concluyentes, sería deseable contar al término de la ejecución con los valores del IPC individuales de cada una. Sin embargo, Multi2Sim sólo devuelve datos globales.

Por tanto, se impone la necesidad de abordar un nuevo desarrollo del código fuente del simulador, a fin de obtener los datos que necesitamos.

Las modificaciones a implementar suponen la creación de un evento periódico que recoja los datos necesarios para el cá-lculo del IPC cada cierto tiempo, de forma análoga a otros eventos que temporales que ya existen en el simulador. Los cambios atañen a los módulos de Multi2Sim relacionados con los contextos de simulación –aplicaciones ejecutadas– y emulación de la arquitectura x86, por lo que los ficheros a alterar se encuentran en la ruta /src/arch/x86/emu (véase apartado 3.1.1).

En síntesis, la idea es agregar una serie de parámetros a los ficheros de configuración de carga para la arquitectura x86, invocados mediante *--x86-config* en las simulaciones, en función de los cuales se recoja y devuelva información relativa al IPC de cada contexto. Estos parámetros serán:

- *EnableReport*. Si existe y tiene un valor *true*, habilitar la toma de datos.
- *ReportFile*. Ruta absoluta del fichero de salida con la información obtenida.
- *ReportInterval*. Intervalo de toma de datos, expresado en ciclos.

Para conseguirlo, se han realizado las siguientes modificaciones:

loader.h. Este fichero contiene las cabeceras que se cargan a memoria al leer los ficheros de configuración relativos al contexto. Declaramos una variable tipo *FILE* para acceder al fichero físico donde se volcarán las estadísticas de salida, y otra tipo *long long* que supondrá el intervalo en ciclos entre las sucesivas tomas de datos:

```
FILE *ipc_report_file;
long long ipc_report_interval;
```

context.h. En el fichero de cabeceras de datos de contextos, declaramos una clase tipo *struct* denominada *x86_ctx_report_stack_t*, que recogerá los datos que necesitamos conocer para el cálculo del IPC (identificador del proceso y contadores de instrucciones y ciclos):

```
struct x86_ctx_report_stack_t {
        int pid;
        long long inst_count;
        long long last_cycle;
}
```

Además, declaramos sendas funciones para programar y manejar los eventos que recogerán los datos:

```
void X86ContextIPCReportSchedule(X86Context *self);
void X86ContextIPCReportHandler(int event, void *data);
```

context.c. En el fichero de implementaciones del contexto de aplicación, creamos el evento EV_X86_CTX_IPC_REPORT y lo registramos. Se imprime en el fichero de salida las cabeceras, se programa el evento para lanzarlo conforme al intervalo establecido.

Se declara una cadena de cabecera para el informe de salida:

```
static char *help_x86_ctx_ipc_report =
```
(El contenido se omite por motivos de espacio)

Implementamos el método correspondiente al evento periódico de recogida y actualización de datos:

```
void X86ContextIPCReportSchedule(X86Context *self)
{
        struct x86_ctx_report_stack_t *stack;
        FILE *f = self->loader->ipc_report_file;

        EV_X86_CTX_IPC_REPORT = esim_register_event_with_name
                (X86ContextIPCReportHandler,
                arch_x86->timing->frequency_domain,"x86_ctx_ipc_report");

        /* Crear Nuevo stack */
        stack = xcalloc(1, sizeof(struct x86_ctx_report_stack_t));

        /* Inicializar */
        assert(self->loader->ipc_report_file);
        assert(self->loader->ipc_report_interval > 0);
        stack->pid = self->pid;

        /* Imprimir cabecera */
        fprintf(f, "%s", help_x86_ctx_ipc_report);
        fprintf(f, "%s\t%s\t%s\t%s\t%s\n", "cycle", "inst", "inst-int",
                "ipc-glob", "ipc-int");

        self->ipc_report_stack = stack;

        /* Programar evento inicial */
        esim_schedule_event(EV_X86_CTX_IPC_REPORT,
                                stack, self->loader->ipc_report_interval);
}
```

Y finalmente desarrollamos el manejador del evento:

```
void X86ContextIPCReportHandler(int event, void *data)
{
	struct x86_ctx_report_stack_t *stack = data;
	long long inst_count;
	double ipc_interval;
	double ipc_global;
	long long L2_accesses_interval;
	long long L2_hits_interval;

	/* Obtener context. Si no existe, no se programan más eventos */
	my_ctx = X86EmuGetContext(x86_emu, stack->pid);
	if (!my_ctx || X86ContextGetState(my_ctx, X86ContextFinished)
			|| esim_finish) return;

	/* No actualizar estadisticas si ha alcanzado max de instrucciones */
	inst_count = my_ctx->inst_count - stack->inst_count;
	ipc_global = esim_cycle() ? (double)
		my_ctx->inst_count / esim_cycle() : 0.0;
	ipc_interval = (double) inst_count / (esim_cycle()
			- stack->last_cycle);

	/* Volcar estadisticas */
	fprintf(my_ctx->loader->ipc_report_file,
		"%10lld\t%8lld\t%8lld\t%10.4f\t%10.4f\n", esim_cycle(),
		my_ctx->inst_count, inst_count, ipc_global, ipc_interval);

	stack->inst_count = my_ctx->inst_count;
	stack->last_cycle = esim_cycle();

	/* Programar el siguiente evento */
	esim_schedule_event(event, stack,
				my_ctx->loader->ipc_report_interval);
}
```

emu.c. Este fichero implementa la emulación principal de la arquitectura x86. Básicamente, aquí se leen los parámetros introducidos en el fichero de contexto, luego si la evaluación booleana de la variable *EnableReport* devuelve un valor verdadero, se instancia el fichero de salida expresado por *ReportFile*, y se programan los eventos definidos anteriormente con el intervalo establecido en *ReportInterval*:

```
snprintf(default_report_file_name, MAX_STRING_SIZE,
        "ctx%d.interval.report", ctx->pid);
enable_report = config_read_bool(config, section, "EnableReport", 0);
ipc_report_file_name = config_read_string(config, section,
                                "ReportFile", default_report_file_name);
loader->ipc_report_interval = config_read_llint(config, section,
                                "ReportInterval", 50000);
if (enable_report)
{
        loader->ipc_report_file = file_open_for_write(ipc_report_file_name);
        if (!loader->ipc_report_file)
        fatal("%s: cannot open interval report file", ipc_report_file_name);
        if (loader->ipc_report_interval < 1)
        fatal("%s: invalid value for 'ReportInterval'", config_file_name);
        X86ContextIPCReportSchedule(ctx);
}
```

m2s.h. Por último, ha sido necesario *crear* este fichero en el directorio raíz /src para permitir a *emu* acceder directamente a las variables de la arquitectura (en caso contrario, el simulador presenta múltiples fallos durante el proceso de compilación):

```
#ifndef M2S_H
#define M2S_H

#include <arch/common/arch.h>

extern struct arch_t *arch_arm;
extern struct arch_t *arch_evergreen;
extern struct arch_t *arch_fermi;
extern struct arch_t *arch_mips;
extern struct arch_t *arch_southern_islands;
extern struct arch_t *arch_x86;

#endif /* M2S_H */
```

4.3.3 Modificaciones en Multi2Sim. Accesos por aplicación

Para conseguir un análisis en profundidad de las prestaciones individuales de cada aplicación en un entorno con varios núcleos, la modificación abordada para obtener el IPC individualizado se antoja insuficiente. Por ello, se decide acometer una segunda ampliación con vistas a este experimento, conducente a obtener además la cantidad de aciertos y fallos a cada módulo de cache efectuado por las distintas aplicaciones.

La implementación del cálculo del IPC individual de cada aplicación ha supuesto una considerable complejidad, tal y como el lector habrá podido apreciar en el apartado 4.3.2. Sin embargo, este desarrollo ha sido hasta cierto punto sencillo, toda vez que los módulos de código fuente afectados se circunscriben al área de Multi2Sim relacionada con el contexto de simulación.

Sin embargo, como se indicó al describir la estructura del código fuente de Multi2Sim y posteriormente se estudió durante las modificaciones efectuadas para la obtención de los accesos por cada vía de la cache (apartados 3.1.1 y 4.1.1, respectivamente), las clases de datos que afectan a la jerarquía de memoria se aglutinan en el módulo *mem-system*. Como los datos estadísticos que se recogen en él son globales, necesitaremos propagar los datos que permitan identificar el núcleo de ejecución que realiza el acceso desde las clases relativas al contexto –donde disponemos de la capacidad de identificarlo– hasta las clases del sistema de memoria donde se producen los aciertos y fallos en la memoria cache .

Para aprovechar parte del trabajo ya realizado, se reutilizará el código desarrollado en el apartado anterior para la impresión de los resultados.

A continuación se resumen las modificaciones realizadas en el código fuente. Debido a su extensión, se ha decidido resumir la información mostrada, mostrando el código fuente implementado sólo cuando su importancia así lo aconseja:

x86/emu/context.c. A fin de reutilizar las utilidades desarrolladas durante el experimento anterior, se declaran aquí las variables estadísticas para permitir imprimirlas en los mismos informes de salida. En este caso, se declaran dos arrays globales de tipo *long long* bajo el epígrafe *global variables* del código fuente:

```
long long *l2_hits;
long long *l2_misses;
```

Además, se inicializan ambas en la función *X86ContextIPCReportSchedule*, con tantas celdas como número de nodos tenemos, obtenido como el producto de las variables global de Multi2Sim de número de núcleos por CPU y número de hilos por CPU:

```
int num_nodes = x86_cpu_num_cores * x86_cpu_num_threads;
l2_hits = xcalloc(num_nodes, sizeof(long long));
l2_misses = xcalloc(num_nodes, sizeof(long long));
```

Para terminar, en la función *X86ContextIPCReportHandler*, imprimiremos estas estadísticas en el mismo fichero marcado por en los ficheros de configuración de contexto:

```
int num_nodes = x86_cpu_num_cores * x86_cpu_num_threads;
for (i=0; i<num_nodes; i++)
{
        fprintf(my_ctx->loader->ipc_report_file, "Core %d\tHits:
        %lld\tMisses: %lld\n", i, l2_hits[i], l2_misses[i]);
```

```
}
```

mem-system/mod-stack.h. Definimos un nuevo atributo *core* de tipo entero para el tipo *mod_stack_t*, que nos permitirá identificar el núcleo que realiza el acceso a cache.

mem-system/mod-stack.c. En la función *mod_stack_create*, asignamos un valor -1 a la variable *stack* de tipo *mod_stack_t* que resulta de la salida de la función. Dado que los núcleos se identifican en Multi2Sim con valores enteros positivos y cero, el valor negativo por defecto resulta útil para identificar valores erróneos propagados por la jerarquía de cache.

mem-system/mod-access.h. En la declaración de la función *mod_access* incluimos un nuevo parámetro de entrada de tipo entero al que denominamos *core*.

mem-system/mod-access.c. En la implementación de la función *mod_access*, pasamos el valor del parámetro *core* al atributo homónimo del objeto *stack*:

```
stack->core = core;
```

mem-system/nmoesi-protocol.c. En la implementación función del protocolo NMOESI para búsqueda y selección del bloque (*mod_handler_nmoesi_find_and_lock*), referenciamos las variables globales declaradas previamente:

```
extern long long *l2_hits;
extern long long *l2_misses;
```

Dentro de la misma función, en el bloque de código correspondiente al manejo del evento *find_and_lock_port*, encontramos las líneas donde se incrementan los accesos *bloqueantes* a los módulos, esto es, peticiones *up-down* que realiza el procesador, descartando los accesos procedentes de niveles superiores de la jerarquía de memoria:

```
if (stack->hit)
{
        if (strcmp(mod->name, "l2") == 0 && !stack->blocking)
                l2_hits[stack->core]++;
}
else
{
        if (strcmp(mod->name, "l2") == 0 && !stack->blocking)
                l2_misses[stack->core]++;
}
```

Por último, en el evento correspondiente a *evictions*, propagamos el identificador de núcleo al crear nuevas pilas de peticiones:

```
new_stack->core = stack->core;
```

mem-system/command.c, **prefectcher.c**. Inclusión del parámetro *core* en las llamadas a la función *mod_access*. Valor por defecto (-1).

x86/timing/fetch.c. En la implementación de la función *X86ThreadFetch*, para el estado correspondiente del ciclo de instrucción, inicializamos el valor del parámetro *core* de *mod_access* pasándole el identificador del núcleo, contenido en el atributo *id_in_cpu* del objeto *self* de tipo *X86_Thread*:

```
self->fetch_access = mod_access(self->inst_mod, mod_access_load,
                        phy_addr,NULL, NULL, NULL, NULL, self->id_in_cpu);
```

El parámetro *id_in_cpu* identifica cada núcleo de forma unívoca en el sistema, con un valor entero que coincide con el identificador de contexto, lo que nos permitirá identificar la aplicación en ejecución.

x86/timing/issue.c. De forma análoga, para el estado *issue* realizaremos la inicialización del identificador del núcleo. En la función *X86ThreadIssueSQ* inicializamos el valor del parámetro *core* de *mod_access* pasándole el identificador del núcleo *stores*:

```
mod_access(self->data_mod, mod_access_store, store->phy_addr, NULL,
                core->event_queue, store, client_info, self->id_in_cpu);
```

evergreen/timing, **mips/timing**, **etc**. Todas las arquitecturas soportadas por el simulador Multi2Sim utilizan los ficheros de la jerarquía de memoria, por lo que en todas existen llamadas a la función *mod_access* que hemos modificado. Dado que la experimentación con arquitecturas diferentes de x86 queda fuera de los objetivos de este proyecto, incluimos un simple valor -1 como parámetro en todas esas llamadas a fin de evitar los errores de compilación.

4.3.4 Resultados de la simulación

MPKI

Comenzaremos el análisis de los datos obtenidos, estudiando el comportamiento del índice MPKI (*Misses Per Kilo-Instruction*) de cada una de las mezclas modeladas.

Dado que el MPKI de un módulo de memoria dado indica el promedio de fallos de acceso por cada mil instrucciones ejecutadas, su comportamiento aporta información valiosa acerca del rendimiento del sistema, en particular en los aspectos referentes al sistema de memoria. Hay que señalar que, dado que a mayor cantidad de fallos de acceso en cualquier nivel de la jerarquía de cache, menor rendimiento experimentará el sistema –debido al aumento de accesos a niveles superiores de la jerarquía, con mayores latencias de acceso–, el índice MPKI es un factor a minimizar para mejorar el rendimiento general del procesador.

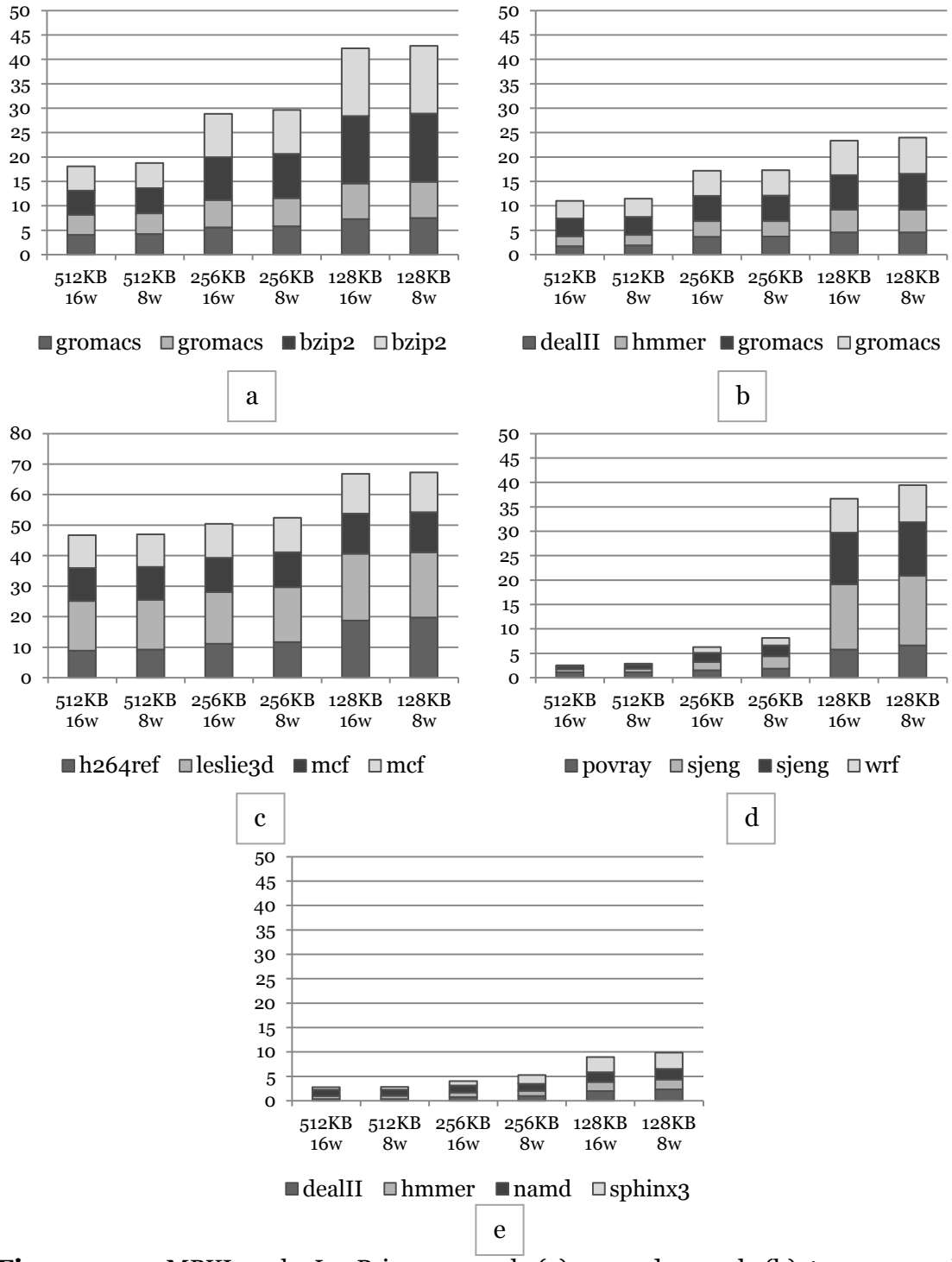

Figura 4.3.1: MPKI, cache L2. Primera mezcla (a), segunda mezcla (b), tercera mezcla (c), cuarta mezcla (d), quinta mezcla (e).

La figura 4.3.1 muestra el índice MPKI (*Misses Per Kilo-Instruction*) correspondiente a la cache de nivel 2, para cada uno de los *benchmarks* de las distintas mezclas. Cada gráfica muestra el comportamiento del MPKI en cada geometría de cache L2 analizada, para la mezcla en cuestión. Además, en cada gráfica se han agrupado los valores del MPKI de los cuatro contextos de las ejecuciones, para mostrar también el MPKI total acumulado en cada simulación. Se ha tratado de unificar la escala vertical para una mejor comparación de los valores mostrados.

Las gráficas de la figura 4.3.1 muestran un progresivo aumento del MPKI individual de cada aplicación de cada mezcla, conforme disminuye el tamaño de la memoria cache de nivel 2, así como un incremento paralelo de inferior magnitud al disminuir a la mitad el número de vías por conjunto, manteniendo fija la capacidad del módulo de memoria. Como sendos incrementos del MPKI se suceden en la totalidad de las aplicaciones, el valor del MPKI acumulado, representado en las gráficas por la altura total de cada columna, aumenta del mismo modo.

Tal y como se ha señalado al comienzo de este apartado, el MPKI afecta de forma inversa al rendimiento. A mayor valor del MPKI, mayor número de fallos de acceso, lo que implica una mayor penalización en el tiempo de ejecución al aumentar los accesos a niveles superiores de la jerarquía de memoria, con latencias más altas, para encontrar y reemplazar los bloques solicitados por las aplicaciones. Por tanto, y a la vista de la figura 4.3.1, el tamaño de la memoria cache –en este caso, L2–, juega un papel fundamental en las prestaciones del procesador. Además, a tenor de lo observado, para un tamaño de memoria cache dado, un menor número de vías implica también menores prestaciones.

Comparando los datos de las gráficas *a* y *b* de la figura 4.3.1, correspondientes a la primera y segunda mezclas, se observa un valor global del MPKI considerablemente superior en la primera de ellas, consecuencia del bajo MPKI obtenido por las aplicaciones dealII y hmmer, en relación a bzip2 (ambas mezclas ejecutan la aplicación gromacs en dos de los núcleos). Nótese que la diferencia existente en el MPKI de estas cargas en el procesador de un núcleo (véanse los resultados del estudio de caracterización de la carga, apartado 4.1.3) se ve acentuada al ejecutarse en un entorno multinúcleo, con más aplicaciones compartiendo la memoria cache de nivel 2. Por otro lado, los valores individuales del MPKI de gromacs (filas superiores de cada columna en la gráfica *b*) son inferiores a los obtenidos por el mismo *benchmark* en la primera mezcla (filas inferiores en la gráfica *a*), lo que constituye una muestra de cómo la naturaleza y comportamiento de las aplicaciones ejecutadas en cada núcleo del procesador afecta al rendimiento de las demás aplicaciones. Así, el menor número de fallos de acceso a L2 de dealII y hmmer, con respecto a bzip2, implica un menor número de reemplazos de bloques en la cache, lo que a su vez provoca un menor número de fallos en el resto de contextos, al mantenerse con mayor probabilidad en la memoria los bloques que han sido referenciados por ellos (en este caso, los contextos que ejecutan gromacs).

Por su parte, en la gráfica *c* de la figura 4.3.1, correspondiente a la tercera mezcla, se ha ampliado la escala vertical, al obtenerse valores del MPKI claramente superiores al resto de mezclas, consecuencia de los altos resultados obtenidos por los *benchmarks* leslie3d y mcf. Cabe señalar que la proporción existente entre los valores individuales del MPKI responde a la clasificación realizada en el estudio de caracterización de la carga (véase apartado 4.1.3). Así, la aplicación h264ref (penalización *baja* por MPKI de nivel 2) se ve menos afectada en sus prestaciones ante la presencia de una memoria cache L2 compartida que leslie3d y mcf.

En la gráfica *d* de la figura 4.3.1 (cuarta mezcla), destacan los bajos valores del MPKI –individuales y globales– para las geometrías de cache de 512KB y 256KB, y el considerable incremento de estos valores para las dos geometrías de 128KB.

En cuanto a la gráfica *e* de la figura 4.3.1 (quinta mezcla de cargas), cabe destacar a su vez los bajos valores globales del MPKI obtenidos, inferiores al resto de mezclas, consecuencia a su vez de la naturaleza de las aplicaciones utilizadas.

Hit Ratio

Continuaremos el estudio de los resultados obtenidos, analizando la tasa de acierto o *HitRatio* de la memoria cache L2.

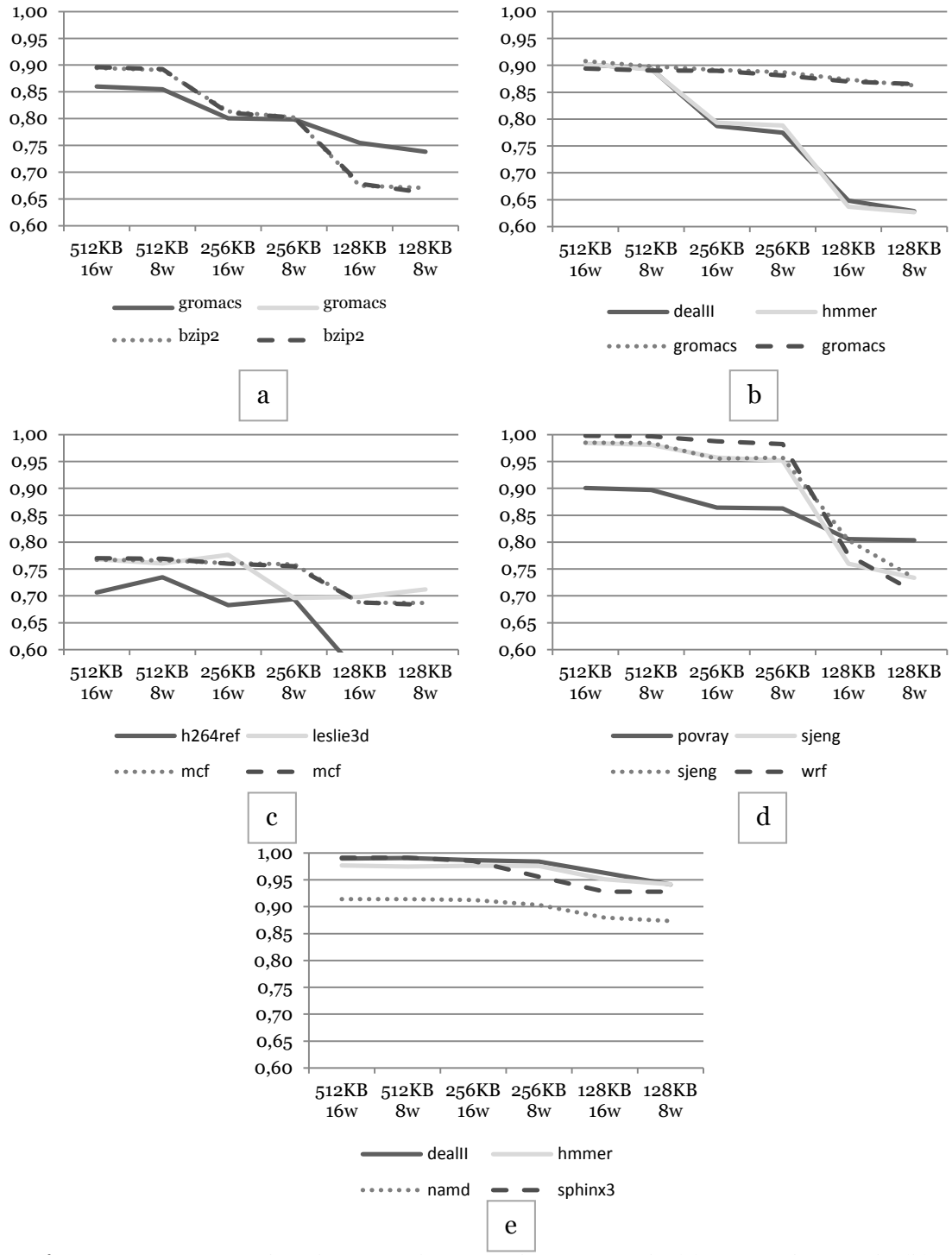

Figura 4.3.2: Tasa de acierto, cache L2. Primera mezcla (a), segunda mezcla (b), tercera mezcla (c), cuarta mezcla (d), quinta mezcla (e).

La figura 4.3.2 muestra la evolución del *HitRatio* de los distintos *benchmarks* de las mezclas estudiadas. Las gráficas muestran una distribución de los valores generalmente escalonada, con pronunciados descensos al disminuir el tamaño de la memoria cache de nivel 2. Igualmente, se aprecia un descenso de menor magnitud en la tasa de aciertos al reducir, para un mismo tamaño de cache, el número de vías por conjunto. Dado que la tasa de aciertos está directamente relacionada con las prestaciones del sistema de memoria y el procesador en general (a mayor tasa de aciertos, menor número de fallos y, por tanto, menor penalización por accesos a niveles superiores de la jerarquía de memoria), se infiere que el tamaño y asociatividad de la memoria cache L2 compartida supone un fuerte impacto sobre las prestaciones.

La gráfica *a* de la figura 4.3.2 muestra la distribución del *HitRatio* para la primera mezcla. Obsérvese que los dos primeros contextos, correspondientes a gromacs, tienen valores prácticamente iguales, por lo que se superponen en la gráfica; los valores de bzip2, sin embargo, difieren algo más. Puede medirse la disminución de las prestaciones al reducir el tamaño y asociatividad de la memoria cache L2. Así, de un valor superior a 0.85 para gromacs y bzip2 en la cache de 512KB y 16 vías, la tasa de aciertos disminuye hasta valores inferiores a 0.75 (gromacs) o ligeramente superiores a 0.65 (bzip2) en la cache de 128KB y 8 vías.

La gráfica *b* muestra la distribución de del *HitRatio* para la segunda mezcla. Mientras que los *benchmarks* dealII y hmmer presentan una evolución escalonada análoga a la observada para la primera mezcla (gráfica *a*), gromacs experimenta un descenso aproximadamente lineal, frente a la distribución escalonada de la primera mezcla. El dispar comportamiento obedece a la perturbación de las prestaciones locales de cada aplicación, al competir por la cache L2 compartida, en base a la naturaleza del resto de aplicaciones.

La gráfica *c* de la figura 4.3.2 muestra la evolución del *HitRatio* de los distintos *benchmarks* de la tercera mezcla. La figura muestra, además del esperado descenso de la tasa de aciertos al disminuir el tamaño de la cache L2, un inesperado aumento –de menor magnitud absoluta que el descenso general–, al disminuir el número de vías para h264ref y leslie3d. Obsérvese que los valores de h264ref para las geometrías de cache con 128KB son menores que la escala inferior de la gráfica (se ha mantenido la escala vertical para coincidir con el resto de gráficas de la figura 4.3.2). En particular, los valores de la tasa de aciertos alcanzados por h264ref en estas dos geometrías son de 0.4732 y 0.4742, correspondientes a la cache L2 de 128KB y 16 vías y a la cache L2 de 128KB y 8 vías, respectivamente.

La gráfica *d* muestra la evolución del *HitRatio* de los distintos *benchmarks* de la cuarta mezcla. La gráfica muestra la distribución escalonada descendente observada en el resto de mezclas de cargas, con una acuciada diferencia entre los valores de las geometrías de 512KB y 256KB por un lado, y las geometrías de 128KB por otro. Especialmente llamativo es el caso del *benchmark* wrf, con valores de tasa de aciertos cercanos a 1.00 para las caches de 512KB y 256KB, que desciende hasta 0.77 en la cache L2 de 128KB y 16 vías, y 0.71 para la cache de 128KB y 8 vías. De la misma forma que se ha ido observando en el resto de mezclas –excepción hecha de los *benchmarks* h264ref y leslie3d en la tercera mezcla–, las geometrías de cache con 16 vías por conjunto

obtienen valores de tasa de aciertos superiores a sus contrapartidas con 8 vías por conjunto.

Por último, la gráfica *d* de la figura 4.3.2 muestra la evolución del *HitRatio* de los distintos *benchmarks* de la quinta mezcla. La gráfica muestra un descenso escalonado, igual que lo observado para el resto de mezclas. En este caso, la pérdida de prestaciones, en cuanto a la tasa de aciertos, es inferior porcentualmente al del resto de mezclas, siendo la pérdida total entre los dos extremos de la gráfica (medida por la diferencia entre el *HitRatio* de la geometría de cache de 512KB y 16 vías y el *HitRatio* de la geometría de 128KB y 8 vías cercana al 5% para los cuatro *benchmarks* que componen la mezcla.

IPC

Para terminar el análisis de los datos obtenidos, examinaremos el comportamiento del IPC (*Instructions Per Cycle*), aprovechando la información suministrada ahora por el simulador tras la modificación efectuada para la obtención del IPC por aplicación.

Cabe recordar que para el estudio nos basamos en el cálculo del IPC tomando como base el número de microinstrucciones (internas de la implementación de la arquitectura x86 del simulador) que han pasado por la fase de *commit*. Este índice IPC se obtiene en base a la fórmula:

$$IPC = \frac{\mu Instrucciones}{Ciclos}$$

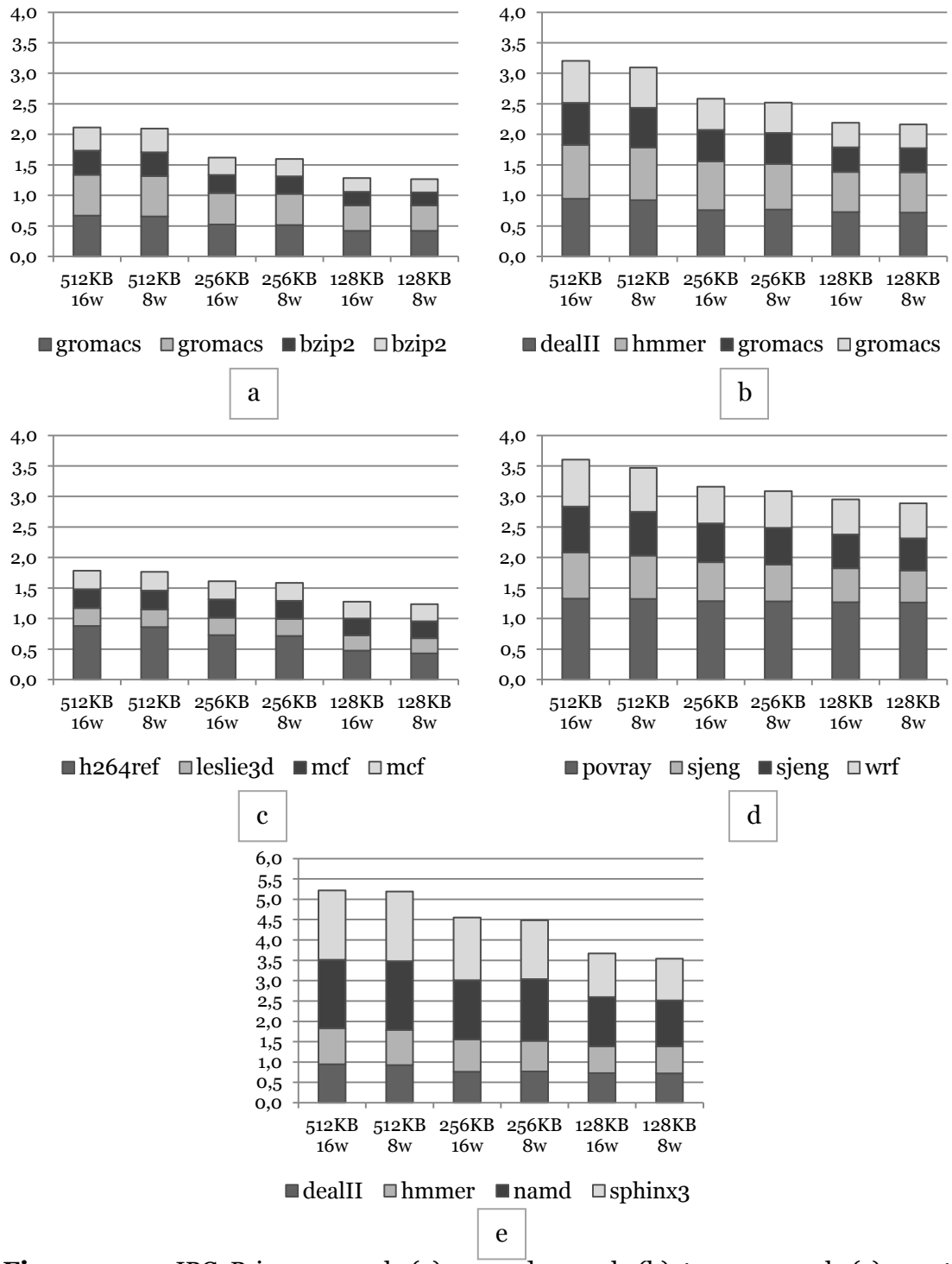

Figura 4.3.3: IPC. Primera mezcla (a), segunda mezcla (b), tercera mezcla (c), cuarta mezcla (d), quinta mezcla (e).

La figura 4.3.3 muestra el IPC individual de cada aplicación, y el IPC global del procesador de cuatro núcleos (valor apilado de cada columna), para cada de las mezclas estudiadas, con cada geometría de cache L2. Se observa claramente el descenso del IPC global conforme reducimos el tamaño y número de vías de la cache, consecuencia de la multiplicación de fallos por conflictos de acceso entre las aplicaciones al nivel compartido de memoria cache.

A la vista del comportamiento individual de cada aplicación, podemos observar igualmente una pérdida de prestaciones, en forma de descenso de su propio IPC, si lo comparamos con los resultados obtenidos por las mismas aplicaciones individualmente a igual tamaño y geometría de la cache de nivel 2 (véanse las gráficas de la fig. 4.2.9 en el apartado 4.2.1, correspondientes a los resultados del experimento de impacto de la cache L2). Este descenso del valor del IPC se acentúa a menor tamaño de la memoria cache de nivel 2 y varía en función de la aplicación.

Analizando las distintas mezclas por separado, la gráfica *a* muestra el IPC, tomando como base el número de microinstrucciones que han realizado la fase de *commit* de la primera de las mezclas. En la gráfica, el *benchmark* gromacs (filas inferiores de cada columna), pasa de un valor del IPC de 0.6716, para la cache de 512KB y 16w con un solo núcleo, a 0.6688 para la misma cache L2 en multinúcleo, lo que supone una pérdida inferior al 1%, mientras que el mismo *benchmark* pasa de un IPC de 0.6259 (cache L2 128KB, 8 vías con un solo núcleo) a 0.4206 para la misma geometría en multinúcleo, lo que supone una pérdida de prestaciones del 33%. De igual forma, mientras que la disminución de prestaciones entre los dos extremos de las geometrías analizadas era tan solo del 6.5% en simulación con un solo núcleo, la diferencia entre el IPC obtenido por gromacs en multinúcleo para la cache de 512KB y 16 vías y la cache de 128KB y 8 vías, arroja una pérdida de rendimiento del 37.2%. Por su parte, el *benchmark* bzip2 (filas superiores de cada columna), desciende de un valor del IPC de 0.4209, para la cache de 512KB y 16w con un solo núcleo, a 0.3937 para la misma cache L2 en multinúcleo, lo que supone una pérdida del 6.5%, mientras que el mismo *benchmark* pasa de un IPC de 0.3879 (cache L2 128KB, 8 vías con un solo núcleo) a 0.2142 para la misma geometría en multinúcleo, lo que implica una pérdida de rendimiento del 45%. De igual forma, mientras que el descenso de las prestaciones entre los dos extremos de las geometrías analizadas era tan solo del 7.9% en simulación con un solo núcleo, la diferencia entre el IPC obtenido por bzip2 en multinúcleo para la cache de 512KB y 16 vías y la cache de 128KB y 8 vías, arroja una pérdida en el rendimiento también del 45%. La considerable pérdida de prestaciones experimentada en ambas aplicaciones está en consonancia con la predicción efectuada al confeccionar la primera mezcla, al tratarse de dos *benchmarks* que *a priori* habían de experimentar serias penalizaciones al competir por las vías de la cache L2.

La gráfica *b* de la figura 4.3.3 muestra el IPC, tomando como base el número de microinstrucciones que han realizado la fase de *commit* de la segunda mezcla. Se aprecia claramente, en comparación con la gráfica *a*, el superior rendimiento conjunto de la segunda mezcla con respecto a la primera (obsérvense los valores globales del IPC, que para la cache de 512KB y 16 vías es de 3.2072 frente al IPC de 2.1127 de la primera mezcla, un 51.8% superior, y para la cache de 128KB y 8 vías es de 2.1650 frente a 1.2663, un 71% superior). Esta diferencia del IPC global se explica por las diferentes prestaciones de cada aplicación. Analizando su comportamiento con un solo núcleo (véase apartado 4.1.3 para los resultados del estudio de caracterización), se aprecia cómo, a igualdad de condiciones en cuanto a geometría de cache se refiere, los *benchmarks* dealII y hmmer obtienen un IPC superior a bzip2, concretamente 0.9464 y 1.028 frente a 0.4209 (evidentemente gromacs no afecta a la diferencia de rendimiento global, ya que se ejecuta dos veces en ambas mezclas).

En lo que respecta al comportamiento individual de las aplicaciones de la segunda mezcla, el *benchmark* dealII (fila inferior de cada columna de la gráfica *b*), pasa de un valor del IPC de 0.9515, para la cache de 512KB y 16w con un solo núcleo, a 0.9468 para la misma cache L2 en multinúcleo, lo que supone un descenso aproximado del 0.5%, mientras que el mismo *benchmark* pasa de un IPC de 0.9086 (cache L2 128KB, 8 vías con un solo núcleo) a 0.7195 para la misma geometría en multinúcleo, lo que implica una pérdida de prestaciones del 21%. De igual forma, mientras que la pérdida de rendimiento entre los dos extremos de las geometrías analizadas era tan solo del 4.5% en simulación con un solo núcleo, la diferencia entre el IPC obtenido por dealII en multinúcleo para la cache de 512KB y 16 vías y la cache de 128KB y 8 vías, arroja una pérdida de rendimiento del 24%. Por su parte, el *benchmark* hmmer (segunda fila inferior de cada columna de la gráfica *b*), desciende de un valor del IPC de 1.0258, para la cache de 512KB y 16w con un solo núcleo, a 0.8809 para la misma cache L2 en multinúcleo, lo que supone una pérdida del 14.1%, mientras que el mismo *benchmark* pasa de un IPC de 0.9827 (cache L2 128KB, 8 vías con un solo núcleo) a 0.6636 para la misma geometría en multinúcleo, lo que implica una pérdida de rendimiento del 32.5%. De igual forma, mientras que el descenso de las prestaciones entre los dos extremos de las geometrías analizadas era tan solo del 4.2% en simulación con un solo núcleo, la diferencia entre el IPC obtenido por hmmer en multinúcleo para la cache de 512KB y 16 vías y la cache de 128KB y 8 vías, arroja una pérdida en el rendimiento también del 24.7%. Por último, los resultados individuales de gromacs son equiparables a los obtenidos con la primera mezcla. Comparando la pérdida de prestaciones de gromacs (penalización *media* por MPKI en L1 y L2 y penalización *alta* por vías de la cache) frente a la ejecución con un solo núcleo, cifrada en un 1% para la cache de 512KB y 16 vías y en un 33% para la L2 de 128KB y 8 vías, con respecto a dealII (de penalización *alta* en el MPKI de nivel 1 y *media* en nivel 2, y penalización *baja* en cuanto a vías de la cache), del 0.5% y 21%, respectivamente, concuerda con la predicción efectuadas al componer la segunda mezcla, en base a la cual la pérdida de prestaciones de gromacs habría de ser claramente superior a la experimentada por esta otra aplicación. Por el contrario, los resultados obtenidos por hmmer no muestran una gran variación con respecto a gromacs (con pérdidas en el IPC del 14.1% y 32.5%).

La gráfica *c* de la figura 4.3.3 muestra el IPC, tomando como base el número de microinstrucciones que han realizado la fase de *commit* de la tercera mezcla. El análisis individual de cada carga permite observar cómo el *benchmark* h264ref (fila inferior de cada columna de la gráfica *c*), pasa de un valor del IPC de 0.9420, para la cache de 512KB y 16w con un solo núcleo, a 0.8796 para la misma cache L2 en multinúcleo, lo que supone un descenso del 7.6%, mientras que el mismo *benchmark* pasa de un IPC de 0.9339 (cache L2 128KB, 8 vías con un solo núcleo) a 0.4318 para la misma geometría en multinúcleo, lo que implica una pérdida de prestaciones del 54%. De igual forma, mientras que la pérdida de rendimiento entre los dos extremos de las geometrías analizadas era inferior al 1% en simulación con un solo núcleo, la diferencia entre el IPC obtenido por h264ref en multinúcleo para la cache de 512KB y 16 vías y la cache de 128KB y 8 vías, arroja una pérdida del 51%. Por su parte, el *benchmark* leslie3d (segunda fila inferior de cada columna), desciende de un valor del IPC de 0.4066, para la cache de 512KB y 16w con un solo núcleo, a 0.2917 para la misma cache L2 en multinúcleo, lo que supone una pérdida del 28.3%, mientras que el mismo *benchmark* pasa de un IPC de 0.3721 (cache L2 128KB, 8 vías con un solo núcleo) a 0.2484 para la misma geometría en multinúcleo, lo que implica una pérdida de

rendimiento del 32.3%. De igual forma, mientras que el descenso de las prestaciones entre los dos extremos de las geometrías analizadas es tan solo del 8.5% en simulación con un solo núcleo, la diferencia entre el IPC obtenido por leslie3d en multinúcleo para la cache de 512KB y 16 vías y la cache de 128KB y 8 vías, arroja una pérdida en el rendimiento del 15%. Por su parte, el *benchmark* mcf (filas superiores de cada columna), desciende de un valor del IPC de 0.3673, para la cache de 512KB y 16w con un solo núcleo, a 0.3077 para la misma cache L2 en multinúcleo, lo que supone una pérdida del 6.3%, mientras que el mismo *benchmark* pasa de un IPC de 0.3478 (cache L2 128KB, 8 vías con un solo núcleo) a 0.2772 para la misma geometría en multinúcleo, lo que implica una pérdida de rendimiento del 20.3%. De igual forma, mientras que el descenso de las prestaciones entre los dos extremos de las geometrías analizadas era tan solo del 5.3% en simulación con un solo núcleo, la diferencia entre el IPC obtenido por mcf en multinúcleo para la cache de 512KB y 16 vías y la cache de 128KB y 8 vías, arroja una pérdida en el rendimiento también del 10%. Destaca la menor pérdida de prestaciones de mcf (6.3% entre la simulación con 512KB y 16 vías en un solo núcleo y los resultados multinúcleo con la misma geometría de cache L2, y 20.3% con respecto al procesador de un solo núcleo para la cache L2 de 128KB y 8 vías), frente a las pérdidas porcentuales observadas en el IPC de 7.6% y 54%, respectivamente, para h264ref y de 28.3% y 32.3%, respectivamente, para leslie3d, sin embargo, la clasificación de las aplicaciones de esta mezcla, h264ref (de penalización *alta* en el MPKI de nivel 1 y *baja* en nivel 2, y penalización *baja* en cuanto a vías de la cache), leslie3d (MPKI L1 *alta*, MPKI L2 *media* y penalización por vías *alta*) y mcf (MPKI L1 *alta*, MPKI L2 *alta* y penalización por vías de la cache *alta*), indicaban *a priori* una mayor penalización en las prestaciones para mfc. No obstante, todas las aplicaciones han experimentado pérdida de prestaciones al enfrentarse a un entorno de competencia por los recursos de la cache de nivel 2.

La gráfica *d* de la figura 4.3.3 muestra el IPC, tomando como base el número de microinstrucciones que han realizado la fase de *commit* de la cuarta mezcla. El IPC global obtenido es comparable al de la tercera mezcla, con valores ligeramente superiores para esta cuarta mezcla, debidos especialmente a la presencia del *benchmark* povray. Mediante el análisis del comportamiento individual de cada aplicación, observamos que el *benchmark* povray (fila inferior de cada columna), pasa de un valor del IPC de 1,3842, para la cache de 512KB y 16w con un solo núcleo, a 1,3279 para la misma cache L2 en multinúcleo, lo que supone un descenso del 4%, mientras que el mismo *benchmark* pasa de un IPC de 1,3029 (cache L2 128KB, 8 vías con un solo núcleo) a 1,2615 para la misma geometría en multinúcleo, lo que implica una pérdida de prestaciones inferior al 10%. De igual forma, mientras que la pérdida de rendimiento entre los dos extremos de las geometrías analizadas era inferior al 6% en simulación con un solo núcleo, la diferencia entre el IPC obtenido por povray en multinúcleo para la cache de 512KB y 16 vías y la cache de 128KB y 8 vías, arroja una pérdida similar. Por su parte, el *benchmark* sjeng (filas intermedias de cada columna), desciende de un valor del IPC de 0.7723, para la cache de 512KB y 16w con un solo núcleo, a 0.7540 para la misma cache L2 en multinúcleo, lo que supone una pérdida inferior al 3%, mientras que el mismo *benchmark* pasa de un IPC de 0.7343 (cache L2 128KB, 8 vías con un solo núcleo) a 0.5238 para la misma geometría en multinúcleo, lo que implica una pérdida de rendimiento del 28.5%. De igual forma, mientras que el descenso de las prestaciones entre los dos extremos de las geometrías analizadas es tan solo del 5% en simulación con un solo núcleo, la diferencia entre el IPC obtenido por

sjeng en multinúcleo para la cache de 512KB y 16 vías y la cache de 128KB y 8 vías, arroja una pérdida en el rendimiento de aproximadamente 30%. Por su parte, el *benchmark* wrf (fila superior de cada columna), desciende de un valor del IPC de 0.8514, para la cache de 512KB y 16w con un solo núcleo, a 0.7699 para la misma cache L2 en multinúcleo, lo que supone una pérdida del 10%, mientras que el mismo *benchmark* pasa de un IPC de 0.8476 (cache L2 128KB, 8 vías con un solo núcleo) a 0.5755 para la misma geometría en multinúcleo, lo que implica una pérdida de rendimiento del 22.1%. De igual forma, mientras que el descenso de las prestaciones entre los dos extremos de las geometrías analizadas era tan solo del 0.5% en simulación con un solo núcleo, la diferencia entre el IPC obtenido por wrf en multinúcleo para la cache de 512KB y 16 vías y la cache de 128KB y 8 vías, arroja una pérdida en el rendimiento también del 25.3%. Cabe destacar que las tres distintas aplicaciones experimentan una relativamente baja (aunque generalizada) pérdida de prestaciones, conforme a lo predicho al confeccionar la mezcla. Los tres *benchmarks*: povray (de penalización *media* por MPKI de nivel 1 y *baja* en nivel 2, y penalización *baja* en cuanto a vías de la cache), sjeng (MPKI L1 *media*, MPKI L2 *baja* y penalización por vías *baja*) y wrf (MPKI L1 *media*, MPKI L2 *baja* y penalización por vías de la cache *baja*) se ven menos penalizados al competir entre sí por los recursos de la cache de nivel 2 que otras aplicaciones con mayores dependencias (compárese con la pérdida de prestaciones de gromacs, en la primera y segunda mezclas).

Por último, la gráfica *d* de la figura 4.3.3 muestra el IPC, tomando como base el número de microinstrucciones que han realizado la fase de *commit* de la cuarta mezcla. Nótese que el IPC global es considerablemente superior al obtenido por cualquier otra mezcla, hasta el punto de haber sido necesario ampliar la escala del eje vertical (en el esto de gráficas se ha optado por mantener el límite superior de la escala en el valor 4.0 del IPC, para homogeneizarlas con el resto de gráficas del IPC que aparecen en el trabajo). Los altos valores del IPC global se deben principalmente a la presencia de los *benchmarks* de coma flotante namd, y sphinx3, cuyo IPC individualizado en procesador de un solo núcleo y cache de 1MB y 16 vías (véase apartado 4.1.3 para los resultados del estudio de caracterización de las cargas), es de 1.8124 y 1.6169, respectivamente. Mediante el análisis del comportamiento individual de cada aplicación, observamos cómo el *benchmark* dealII (fila inferior de cada columna), pasa de un valor del IPC de 0.9515, para la cache de 512KB y 16w con un solo núcleo, a 0.9468 para la misma cache L2 en multinúcleo, lo que supone un descenso aproximado del 0.5%, mientras que el mismo *benchmark* pasa de un IPC de 0.9086 (cache L2 128KB, 8 vías con un solo núcleo) a 0.7195 para la misma geometría en multinúcleo, lo que implica una pérdida de prestaciones del 21%. De igual forma, mientras que la pérdida de rendimiento entre los dos extremos de las geometrías analizadas era tan solo del 4.5% en simulación con un solo núcleo, la diferencia entre el IPC obtenido por dealII en multinúcleo para la cache de 512KB y 16 vías y la cache de 128KB y 8 vías, arroja una pérdida de rendimiento del 24%. Por su parte, el *benchmark* hmmer (segunda fila inferior de cada columna), desciende de un valor del IPC de 1.0258, para la cache de 512KB y 16w con un solo núcleo, a 0.8809 para la misma cache L2 en multinúcleo, lo que supone una pérdida del 14.1%, mientras que el mismo *benchmark* pasa de un IPC de 0.9827 (cache L2 128KB, 8 vías con un solo núcleo) a 0.6636 para la misma geometría en multinúcleo, lo que implica una pérdida de rendimiento del 32.5%. De igual forma, mientras que el descenso de las prestaciones entre los dos extremos de las geometrías analizadas era tan solo del 4.2% en simulación con un solo núcleo, la diferencia entre el

IPC obtenido por hmmer en multinúcleo para la cache de 512KB y 16 vías y la cache de 128KB y 8 vías, arroja una pérdida en el rendimiento también del 24.7%. Por su parte, el *benchmark* namd (tercera fila inferior de cada columna), desciende de un valor del IPC de 1.7031, para la cache de 512KB y 16w con un solo núcleo, a 1.6843 para la misma cache L2 en multinúcleo, lo que supone una pérdida del 1.2%, mientras que el mismo *benchmark* pasa de un IPC de 1.7019 (cache L2 128KB, 8 vías con un solo núcleo) a 1.1327 para la misma geometría en multinúcleo, lo que implica una pérdida de rendimiento del 33.5%. De igual forma, mientras que el descenso de las prestaciones entre los dos extremos de las geometrías analizadas es ínfimo (inferior al 1‰) en simulación con un solo núcleo, la diferencia entre el IPC obtenido por namd en multinúcleo para la cache de 512KB y 16 vías y la cache de 128KB y 8 vías, arroja una pérdida en el rendimiento del 32.8%. Por último, el *benchmark* sphinx3 (fila superior de cada columna), desciende de un valor del IPC de 1.7022, para la cache de 512KB y 16w con un solo núcleo, a 1.7007 para la misma cache L2 en multinúcleo, lo que supone un descenso aproximado del 0% mientras que el mismo *benchmark* pasa de un IPC de 1.6815 (cache L2 128KB, 8 vías con un solo núcleo) a 1.0230 para la misma geometría en multinúcleo, lo que implica una pérdida de rendimiento del 40%. De igual forma, mientras que el descenso de las prestaciones entre los dos extremos de las geometrías analizadas era tan solo del 2.3% en simulación con un solo núcleo, la diferencia entre el IPC obtenido por sphinx3 en multinúcleo para la cache de 512KB y 16 vías y la cache de 128KB y 8 vías, arroja una pérdida en el rendimiento del 40%. Cabe destacar la escasa pérdida de prestaciones experimentada por las cuatro aplicaciones (compárese de nuevo con el comportamiento de *benchmarks* como gromacs), lo que concuerda con la clasificación de las mismas efectuada a la hora de confeccionar las mezclas. Así, dealII (de penalización *alta* en el MPKI de nivel 1 y *media* en nivel 2, y penalización *baja* en cuanto a vías de la cache), hmmer (MPKI L1 *media*, MPKI L2 *baja* y penalización por vías *baja*), namd (MPKI L1 *baja*, MPKI L2 *baja* y penalización por vías de la cache *baja*) y sphinx3 (MPKI L1 *baja*, MPKI L2 *baja* y penalización por vías de la cache *baja*) se ven menos penalizados al competir por los recursos de la cache L2 que otras aplicaciones.

4.4 Conclusiones

A lo largo de este capítulo hemos llevado a cabo el procedimiento experimental encaminado a obtener resultados que nos permitan alcanzar el objetivo de evaluar la jerarquía de cache en los procesadores multinúcleo.

Inicialmente hemos descrito los dos procesadores modelados para los distintos experimentos: un procesador de un solo núcleo y otro dotado con cuatro núcleos.

Hemos comenzado por realizar un estudio de caracterización de las cargas (apartado 4.1), encaminado a extraer las características y comportamiento de los *benchmarks* de SPEC CPU2006 en un entorno con un solo núcleo, el cual nos ha permitido analizar su naturaleza y prestaciones individuales, y poder así categorizarlas. Previamente, ha sido necesario realizar una modificación del código fuente del simulador Multi2Sim, para obtener así información relativa a la distribución de aciertos entre las vías de la cola LRU de cada módulo de cache.

Posteriormente, se ha realizado un estudio sobre el impacto de la geometría de la cache de nivel 2, llevando a cabo sucesivas simulaciones en un procesador de un solo núcleo variando el tamaño y geometría de la cache L2, analizando después la variación del rendimiento de las cargas de trabajo.

Por último, el experimento principal se ha llevado a cabo en el procesador multinúcleo. Con la información obtenida en el estudio de caracterización de la carga, se realizado una clasificación de las aplicaciones en base a tres criterios cuantitativos: sus valores de MPKI (*Misses Per Kilo-Instruction*) tanto de la cache de nivel 1 como de la de nivel 2, y su necesidad en vías de la cache L2 en base al porcentaje de aciertos en las vías de la cola LRU, conformando después cinco mezclas de aplicaciones. Tras realizar nuevas modificaciones conducentes a obtener nuevas funcionalidades en Multi2Sim, estas cinco se han sometido a una serie de experimentos variando el tamaño y geometría de la cache compartida de nivel 2, utilizando para ello las mismas geometrías que se han empleado en el estudio sobre el impacto de la geometría de cache. Los resultados de esta serie de simulaciones nos han aportado la información necesaria para conseguir el objetivo de evaluación de la jerarquía de cache.

5. Evaluación de los resultados

A lo largo del trabajo, se han elaborado sucesivos experimentos conducentes a evaluar el rendimiento de la jerarquía de cache en procesadores multinúcleo, y analizar el comportamiento correspondiente en los programas que se ejecutan en ellos.

Tras estudiar las prestaciones de cargas reales en un sistema con un solo núcleo, se han sometido a diversas pruebas de estrés en un entorno multinúcleo donde la cache L2 actúa de nexo compartido, y analizado la pérdida de rendimiento que cabe esperar al introducir un elemento común en la jerarquía de cache que implica la aparición de conflictos y la penalización en el acceso a memoria entre los programas en ejecución.

Las pruebas realizadas se revelan concluyentes en cuanto a la confirmación de esta aseveración. La comparativa entre el rendimiento individualizado de las cargas, tomando como principal medida sus índices de IPC y MPKI, y el rendimiento en sistemas multinúcleo, ha revelado una pérdida general en las prestaciones de ejecución individuales de cada programa.

El análisis de los resultados de la evaluación de prestaciones en un entorno multinúcleo (apartado 4.3), ha permitido además establecer de forma porcentual la pérdida sufrida por las distintas aplicaciones al encontrarse en estas condiciones de competencia por los recursos de la cache compartida. Asimismo, se han establecido unas pautas de análisis *a priori* del comportamiento que cada aplicación experimentará en un procesador multinúcleo, a partir del análisis de su rendimiento en un entorno de un solo núcleo, con un grado aceptable de acierto.

A modo de ejemplo, puede compararse la pérdida de prestaciones del *benchmark* gromacs (penalización *media* por MPKI en L1 y L2 y penalización *alta* por vías de la cache) frente a la ejecución con un solo núcleo, cifrada en un 1% para la cache de 512KB y 16 vías y en un 33% para la L2 de 128KB y 8 vías, con respecto a dealII (de penalización *alta* en el MPKI de nivel 1 y *media* en nivel 2, y penalización *baja* en cuanto a vías de la cache), del 0.5% y 21%

Asumiendo que el principal factor de pérdida de prestaciones en la cache se debe a su tasa de acierto, toda vez que cada fallo implica el acceso a la información contenida en los niveles superiores de la jerarquía de memoria, se han identificado dos factores clave en la disminución de esta tasa y, como consecuencia directa, el aumento del número de fallos de acceso a la cache.

Por un lado, la capacidad de los módulos de cache guarda una relación directa con el rendimiento experimentado. A menor tamaño de la cache, menor es la cantidad de información que puede almacenar, y por tanto mayor el número de accesos a los niveles superiores de la jerarquía de memoria.

Por otro lado, y de forma secundaria, el grado de asociatividad de la cache. En computadores multinúcleo, donde existen módulos de cache compartidos entre los distintos núcleos de ejecución, cuando los núcleos ejecutan cargas de trabajo que

distribuyen individualmente sus aciertos entre un gran número de vías, los fallos de acceso a cache pueden implicar el reemplazo de bloques que otras aplicaciones estén accediendo, lo que provoque a su vez fallos adicionales en dichas aplicaciones cuando intenten acceder a información que, en un entorno con un solo núcleo, ya hubiesen estado presentes en la cache.

Además, los resultados obtenidos en el apartado 4.3.4, indican también que el impacto de la geometría de la cache en entornos multinúcleos es considerablemente mayor que en procesadores de un solo núcleo. Así, aplicaciones como bzip2, cuyo rendimiento cae un 7.9% al reducir la cache L2 desde 512KB y 16 vías a 128KB y 8 vías en un procesador con un solo núcleo, experimenta una pérdida de prestaciones del 45% si esta modificación en la geometría de la cache se realiza en un procesador de cuatro núcleos. Otros *benchmarks* obtienen resultados aún peores: h264ref incrementa la pérdida de rendimiento desde solo un 1% (procesador de un núcleo) a un 51% (procesador multinúcleo). La disminución de la cache compartida en un procesador multinúcleo tiene por tanto consecuencias exponenciales en la pérdida de prestaciones individuales de las aplicaciones ejecutadas por cada uno de los núcleos.

Dado que, tal y como se mencionó al principio de este trabajo, la tecnología actual no permite grandes márgenes en cuanto a la capacidad de las memorias cache se refiere -los módulos actuales disponibles en el mercado cuentan con tamaños del orden de unos pocos MBytes-, la correcta selección de la geometría de los distintos niveles de cache se revela como un factor clave en la industria para la obtención de jerarquías de caches eficientes en entornos multinúcleo que son, cabe recordarlo, la base de todos los chips actuales.

A pesar de las limitaciones impuestas por el carácter de un trabajo fin de grado, así como por el margen de tiempo disponible, el estudio ha permitido servir como una valiosa introducción a la investigación en el ámbito de la ingeniería de computadores, aportando además gran cantidad de información para la demostración empírica de los objetivos propuestos.

Este trabajo, además, ha supuesto una introducción práctica a las técnicas y metodologías de la investigación, aplicada a la transferencia de conocimientos para la mejora de los productos y procesos de la industria de la ingeniería de computadores, así como una primera toma de contacto con diversas herramientas y tecnologías de crucial valor tanto para la ingeniería de computadores como para las tecnologías de la información, aportando un plus a los conocimientos adquiridos durante su desarrollo.

Julio Antonio Vivas Vivas

Valencia, agosto de 2014

6. Bibliografía

[1] D. A. Patterson and J. L. Hennessy, Computer Organization and Design: the Hardware-Software Interface, San Francisco: Morgan Kaufmann, 2012.

[2] D. A. Patterson, J. L. Hennessy and K. Asanovic, Computer Architecture: a quantitative approach, Waltham: Morgan Kaufmann, 2012.

[3] V.V.A.A.; Northeastern University, Boston, MA, USA, Universidad Politécnica de Valencia, Spain, "The Multi2Sim Simulation Framework," 2014.

[4] Henning, John L.; SPEC CPU Subcommittee, Sun Microsystems, "SPEC CPU2006 Benchmark Description," *ACM SIGARCH Computer Architecture News,* 2006.

[5] A. Valero, S. Petit, J. Sahuquillo, P. López and J. Duato, "Analyzing the Optimal Ratio of SRAM Banks in Hybrid Caches," *Computer Design (ICCD),* 2012.

[6] J. P. Shen and M. H. Lipasti, Modern processor design: Fundamentals of superscalar processors, Boston: McGraw-Hill Higher Education, 2005.

Apéndices

Apéndice I. Contenido del fichero *--mem-report* de Multi2Sim

Multi2Sim nos devuelve una ingente cantidad de información sobre la simulación efectuada. Para el estudio en que nos encontramos, nos interesa especialmente el contenido del fichero estadístico sobre la jerarquía de memoria, cuya ruta se especifica con el parámetro *--mem-report* al ejecutar el simulador.

A continuación, se muestra toda la información contenida en uno de estos ficheros, para la simulación con el *benchmark* perlbench y una cache de nivel 2 de 512KB, asociativa por conjuntos de 8 vías. Se han omitido los datos estadísticos del tráfico en las redes de interconexión por motivo de espacio y falta de interés para el estudio.

[dl1]

Sets = 128
Assoc = 2
Policy = LRU
BlockSize = 64
Latency = 2
Ports = 2

Accesses = 63616300
Hits = 61748088
Misses = 1868212
HitRatio = 0.9706
Evictions = 1814132
Retries = 109

Reads = 76449516
ReadRetries = 97
BlockingReads = 38127960
NonBlockingReads = 0
ReadHits = 36641078
ReadMisses = 39808438

Writes = 40881316
WriteRetries = 12
BlockingWrites = 25488340
NonBlockingWrites = 0
WriteHits = 25107010
WriteMisses = 15774306

NCWrites = 0
NCWriteRetries = 0
NCBlockingWrites = 0
NCNonBlockingWrites = 0
NCWriteHits = 0
NCWriteMisses = 0
Prefetches = 0
PrefetchAborts = 0
UselessPrefetches = 0

NoRetryAccesses = 63616185
NoRetryHits = 61748088
NoRetryMisses = 1868097
NoRetryHitRatio = 0.9706
NoRetryReads = 38127857
NoRetryReadHits = 36641078
NoRetryReadMisses = 1486779
NoRetryWrites = 25488328
NoRetryWriteHits = 25107010
NoRetryWriteMisses = 381318
NoRetryNCWrites = 0
NoRetryNCWriteHits = 0
NoRetryNCWriteMisses = 0

[il1]

Sets = 128
Assoc = 2
Policy = LRU
BlockSize = 64
Latency = 2
Ports = 2

Accesses = 35065695
Hits = 33391122
Misses = 1674573
HitRatio = 0.9522
Evictions = 1568933
Retries = 177

Reads = 36366454
ReadRetries = 177
BlockingReads = 34960984
NonBlockingReads = 0
ReadHits = 33286411
ReadMisses = 3080043

Writes = 104711
WriteRetries = 0
BlockingWrites = 104711
NonBlockingWrites = 0
WriteHits = 104711
WriteMisses = 0

NCWrites = 0
NCWriteRetries = 0
NCBlockingWrites = 0
NCNonBlockingWrites = 0
NCWriteHits = 0
NCWriteMisses = 0
Prefetches = 0
PrefetchAborts = 0
UselessPrefetches = 0

NoRetryAccesses = 35065509
NoRetryHits = 33391122
NoRetryMisses = 1674387
NoRetryHitRatio = 0.9522
NoRetryReads = 34960798
NoRetryReadHits = 33286411
NoRetryReadMisses = 1674387
NoRetryWrites = 104711
NoRetryWriteHits = 104711
NoRetryWriteMisses = 0
NoRetryNCWrites = 0
NoRetryNCWriteHits = 0
NoRetryNCWriteMisses = 0

[l2]

Sets = 1024
Assoc = 8

```
Policy = LRU
BlockSize = 64
Latency = 6
Ports = 2

Accesses = 8567602
Hits = 6922787
Misses = 1644815
HitRatio = 0.808
Evictions = 133
Retries = 0

Reads = 3159678
ReadRetries = 0
BlockingReads = 0
NonBlockingReads = 3159678
ReadHits = 1819997
ReadMisses = 1339681

Writes = 5407924
WriteRetries = 0
BlockingWrites = 1643486
NonBlockingWrites = 3764438
WriteHits = 5102790
WriteMisses = 305134

NCWrites = 0
NCWriteRetries = 0
NCBlockingWrites = 0
NCNonBlockingWrites = 0
NCWriteHits = 0
NCWriteMisses = 0
Prefetches = 0
PrefetchAborts = 0
UselessPrefetches = 0

NoRetryAccesses = 8567602
NoRetryHits = 6922787
NoRetryMisses = 1644815
NoRetryHitRatio = 0.808
NoRetryReads = 3159678
NoRetryReadHits = 1819997
NoRetryReadMisses = 1339681
NoRetryWrites = 5407924
NoRetryWriteHits = 5102790
NoRetryWriteMisses = 305134
NoRetryNCWrites = 0
NoRetryNCWriteHits = 0
NoRetryNCWriteMisses = 0


[ mod-mm ]
Sets = 128
Assoc = 8
Policy = LRU
BlockSize = 64
Latency = 300
Ports = 2

Accesses = 1644948
Hits = 398
Misses = 1644550
HitRatio = 0.000242
Evictions = 1643526
Retries = 0

Reads = 1339681
ReadRetries = 0
BlockingReads = 0
NonBlockingReads = 1339681
ReadHits = 254
ReadMisses = 1339427

Writes = 305267
WriteRetries = 0
BlockingWrites = 0
NonBlockingWrites = 305267
WriteHits = 144
WriteMisses = 305123

NCWrites = 0
NCWriteRetries = 0
NCBlockingWrites = 0
NCNonBlockingWrites = 0
NCWriteHits = 0
NCWriteMisses = 0
Prefetches = 0
PrefetchAborts = 0
UselessPrefetches = 0

NoRetryAccesses = 1644948
NoRetryHits = 398
NoRetryMisses = 1644550
NoRetryHitRatio = 0.000242
NoRetryReads = 1339681
NoRetryReadHits = 254
NoRetryReadMisses = 1339427
NoRetryWrites = 305267
NoRetryWriteHits = 144
NoRetryWriteMisses = 305123
NoRetryNCWrites = 0
NoRetryNCWriteHits = 0
NoRetryNCWriteMisses = 0
```

Figura ap1: Fichero de salida --mem_report, *benchmark* perlbench.

Los datos expuestos en la figura superior muestran la potencia de Multi2Sim para la evaluación del rendimiento de los sistemas informáticos, concretamente en el ámbito

de los microprocesadores, y permiten comprender el porqué de su aceptación para la investigación y experimentación en el terreno de la ingeniería de computadores.

Como puede apreciarse, Multi2Sim separa los datos estadísticos para cada módulo de la jerarquía de memoria en seis grandes bloques. Comienza describiendo el módulo en sí, tal y como se ha reflejado en el fichero de configuración de memoria, para continuar exponiendo los valores totales de gran cantidad de variables.

El segundo bloque pormenoriza la cantidad total de accesos, aciertos y fallos que se han producido en el módulo durante la ejecución. Este bloque de información es clave para el estudio del comportamiento de la jerarquía de memoria cache que estamos abordando.

El resto de bloques, que contabilizan cantidad y tipo de las diferentes lecturas y escrituras producidas en el módulo, queda fuera del alcance de este trabajo, por lo que sólo se ha incluido aquí como información ilustrativa de las capacidades del simulador.