UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Regulación de la prebúsqueda considerando el ancho de banda de memoria en el IBM POWER8

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Carlos Navarro Serra
Tutor: Julio Sahuquillo Borrás
Cotutor: Salvador Vicente Petit Martí
Director experimental: Josué Feliu Pérez

Curso Julio 2018

Lista de acrónimos

BAPC Bandwidth-Aware Prefetcher Configuration. 5, 37, 38, 41–43, 45, 46, *Glosario:* BAPC

DRAM Dynamic Random Access Memory. 1, 14, 17, *Glosario:* DRAM

DSCR Data Streams Control Register. 17–19, 25, 27, 34, 47

eDRAM Embedded *DRAM*. 14, 17

FDP Feedback Directed Prefetching. 9

IPC Instructions Per Cycle . 3, 5, 6, 11, 23, 27, 32–35, 37, 38, 45, *Glosario:* IPC

LLC Last Level Cache. 1–3, 9, 14, 16, 22, 29, *Glosario:* LLC

MCDRAM Multi-Channel *DRAM*. 14

MPKI Misses Per K-Instruction. 11

PID Process Identifier. 20, 26

PMU Performance Monitoring Unit. 22

ROB Reorder Buffer. 2, *Glosario:* ROB

SMT Simultaneous Multithreading. 10, 17

SRAM Static Random Access Memory. 17

Glosario

array Vector de elementos. 29, 30

BAPC Bandwidth-Aware Prefetcher Configuration es la propuesta desarrollada en este trabajo. 37, 45

bash Terminal de comandos de Linux.. 20, 26, 46

benchmark Aplicación estandarizada usada para evaluar un sistema. 23, 25, 34

DRAM Tecnología de memoria dinámica de acceso aleatorio. En este tipo de memoria es necesario refrescar los datos en un periodo denominado ciclo de refresco. La razón es que están basadas en condensadores y, de no refrescar, los datos se perderían. 1

hardware Parte física de un computador. Los componentes que forman parte de una computadora. 11, 15, 17–19, 22

hit Acierto. En el ámbito de memorias de computadores. Cuando una aplicación solicita un dato y este se encuentra en una memoria caché. 13, 14, 16

kernel de Linux Núcleo de Linux.. 17, 22

LLC Último nivel o nivel más bajo de la jerarquía de memorias caché. 1

load Tipo de instrucción del procesador para la carga de datos. 18, 19, 22

microbenchmark Aplicación que permite la creación de diversos escenarios. Concepto desarrollado con más detalle en el capítulo5. sección5.2. 11, 29–35

miss Fallo. En el ámbito de memorias de computadores. Cuándo una aplicación solicita un dato y este no se encuentra en una memoria caché. Esto genera un acceso a memoria principal o un nivel inferior de la jerarquía. 13, 15

off-chip Referido a los componentes que no están incluidos en el propio chip. 13

pipeline Referencia a la estructura del procesador. 13

prefetch Prebúsqueda. 2, 5, 7, 11, 13–16, 27, 33–35, 37, 38, 42, 45

prefetcher Mecanismo de prebúsqueda. 2–7, 9–11, 15–19, 22, 23, 25, 27, 28, 32–35, 37, 38, 41, 45–47

ROB Estructura interna del procesador. Ésta permite la finalización en orden en un procesador fuera de orden. 2

software Conjunto de instrucciones (programas) que permiten a una computadora la realización de determinadas tareas. 15, 18, 25, 29

speedup Métrica que indica el nivel de ganancia. 15, 23, 38, 42

store Tipo de instrucción del procesador para el almacenamiento de datos. 18, 19

stride Paso. Separación entre diferentes accesos de memoria. 19, 29–31

Resumen

Hoy en día, los procesadores implementan una serie de *prefetchers* a lo largo de la jerarquía de caché del procesador con el objetivo de reducir u ocultar la latencia de los accesos a memoria y, con ello, mejorar las prestaciones del sistema. Puesto que estos *prefetchers* poseen una alta complejidad y su configuración afecta notablemente las prestaciones del sistema, es importante realizar una óptima gestión de estos recursos.

En este trabajo se realiza un estudio del impacto de las diferentes configuraciones del *prefetcher* del **IBM POWER8** en las prestaciones de las aplicaciones (*benchmarks*). El estudio presenta una caracterización de las prestaciones alcanzadas por las aplicaciones en función del ancho de banda de memoria principal consumido. La caracterización cubre tanto la ejecución en solitario de cada una de las aplicaciones como la ejecución concurrente con otras aplicaciones, donde se generan escenarios de alta contención de ancho de banda de memoria principal. Basándonos en los resultados del estudio, se ha propuesto *Bandwidth-Aware Prefetcher Configuration* (*BAPC*). Se trata de una estrategia de prebúsqueda que elige la configuración óptima del *prefetcher* para cada aplicación en ejecución multinúcleo. El objetivo perseguido es realizar una mejor gestión del ancho de banda del sistema, y repartirlo de la manera más adecuada, para mejorar sus prestaciones globales. Los resultados son realmente esperanzadores y se consiguen mejorar las prestaciones en alrededor de un 70 % en algunas aplicaciones y en un 30 % de media en las aplicaciones estudiadas.

Palabras clave: *Prefetch*,*Prefetcher*,Prebúsqueda, IBM POWER8, DSCR, Memoria, Ancho de banda, Caché.

Resum

Hui en dia, els processadors actuals implementen una sèrie de *prefetchers* al llarg de la seua jerarquia de caché del processador amb l'objectiu de reduir o ocultar la latència dels accessos a memòria i, amb això, millorar les prestacions del sistema. Pel fet que aquests *prefetchers* tenen una alta complexitat i la seua configuració altera les prestacions del sistema, és important fer una òptima gestió d'aquests.

En aquest treball s'ha realitzat un estudi de l'impacte de les diferents configuracions del prefetcher del **IBM POWER8** en l'execució de certes aplicacions (*benchmarks*). L'estudi presenta una caracterització de les prestacions en funció de l'ample de banda de memòria principal consumit. La caracterització s'ha realitzat tant en una execució en solitari com en execució concurrent amb altres aplicacions, on es genera situacions d'alta contenció d'amplada de banda de memòria principal. Basant-nos en els resultats de l'estudi, s'ha proposat *Bandwidth-Aware Prefetcher Configuration* (*BAPC*). Es tracta d'una estratègia de prebúsqueda que tria una configuració òptima del *prefetcher* per a cada aplicació en execució multinucli. L'objectiu perseguit és fer un millor ús de l'ample de banda del sistema, i repartir-ho d'una forma més adequada, per a millorar les seues prestacions globals. Els resultats són realment esperançadors i mostren beneficis en les prestacions de fins a un 70 % en algunes aplicacions i d'un 30 % de mitja en les aplicacions estudiades.

Paraules clau: *Prefetch*,*Prefetcher*, IBM POWER8, DSCR, Memória, Ample de banda, Caché.

Abstract

Nowadays current processors implements a range of *prefetchers* along its processor memory cache hierarchy with the aim of decreasing or hiding the memory accesses lantency and consecuently increasing the system's performance. Due to the complexity of those *prefetchers* and because their configuration alters the system's performance, it is crucial to make an optimal management of them.

Therefore, in this work a study of the impact of the diferent *prefetch* configurations in the **IBM POWER8** on several aplications (*benchmarks*) has been carried out. This study shows a characterization of the reached performance of the aplications depending on main memory bandwidth consumption. The characterization covers not only isolated execution but also concurrent execution with other aplications, where a high memory bandwidth contention is present. Based on the results of this study, we proposed *Bandwidth-Aware Prefetcher Configuration* (*BAPC*). It is a prefetch strategy that determines the optimal *prefetcher* configuration for each aplication in multicore execution. The aim of this strategy is to make a better use of the system bandwith and also its appropriate ditribution. The encouraging results show that the performance of some aplications is increased up to 70 % and 30 % in average with the studied aplications.

Key words: *Prefetch*, *Prefetcher*, IBM POWER8, DSCR, Memory, Bandwidth, Cache.

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

1.1 Descripción del problema

El funcionamiento básico de un procesador clásico consiste en la lectura de datos de memoria, operar sobre ellos, y una vez realizadas las operaciones, guardar los resultados en la memoria. Esto nos lleva a dos componentes básicos, el procesador en sí y la memoria.

Durante las últimas décadas, el desarrollo de los procesadores y el de memorias ha seguido caminos diferentes. Esta división en dos campos ha provocado lo que se conoce como la brecha de prestaciones entre procesador y memoria. Esta brecha se ha traducido en que, mientras el procesador incrementa sus prestaciones entorno a un 60 % por año, la mejora en el tiempo de acceso a las memorias del tipo *Dynamic Random Access Memory (DRAM)* no llega a un 10 % [15]. Además, puesto que la escala de integración ha ido aumentando a lo largo de los años -–cada dos años se duplica el número de transistores en un microprocesador, como predijo la Ley de Moore [9]-–, se ha incrementado la capacidad de añadir un mayor número de transistores dentro del procesador. Por ello, uno de los principales retos en el diseño desde los primeros computadores ha sido utilizar estos transistores adicionales para reducir y ocultar las latencias en los accesos a memoria principal.

Una primera forma de atacar el problema de las latencias de acceso a la memoria principal fue la inclusión de memorias caché. Las memorias caché son estructuras relativamente pequeñas que almacenan datos y/o instrucciones. Estas se sitúan entre el procesador y la memoria principal. Su pequeño tamaño permite que su tiempo de acceso sea muy reducido en comparación con la memoria principal. Estas memorias permiten ocultar en gran medida la latencia de acceso a memoria gracias a la localidad (tanto espacial como temporal) que exhiben los dados e instrucciones.

La creciente diferencia entre la memoria principal y la memoria caché hizo que los procesadores empezasen a incorporar jerarquías de caché. Es decir, múltiples niveles de caché -– tres en la mayoría de los procesadores multinúcleo actuales comerciales —. La alta localidad presentada por la mayoría de las aplicaciones se observa en que la mayor parte de los accesos encuentran el dato accedido en la memoria caché de nivel 1 (L1), y un alto porcentaje en la de nivel 2 (L2). El resto de los datos se encuentran muy probablemente en la caché de último nivel; L3 o *Last Level Cache (LLC)* en la mayoría de los procesadores. A pesar de esto, si un dato no se encuentra dentro de la jerarquía, el procesador debe generar una petición de acceso a memoria principal, cuya latencia supone un gran coste para las prestaciones de la aplicación.

Como se ha comentado, la escala de integración se ha ido aumentando, lo que ha posibilitado el incremento de los tamaños de las memorias caché y en algunos casos generar una jerarquía de memoria más profunda; lo que, en los peores, casos puede hacer que esta latencia que tratamos de mitigar sea aún peor.

Desafortunadamente, la jerarquía de caché por sí sola no es suficiente para ocultar la latencia de acceso a memoria completamente, lo cual puede repercutir negativamente en las prestaciones de muchas aplicaciones. Cuando se accede a un dato que no está almacenado en las memorias caché, la latencia de acceso a la memoria principal, que supera el un centenar de ciclos del procesador, bloquea el *Reorder Buffer (ROB)* durante decenas de ciclos. Cuando se accede frecuentemente a memoria principal, esta situación se convierte en un importante cuello de botella para las prestaciones del sistema.

La alta latencia de acceso a memoria principal viene dada por dos razones principales. En primer lugar, como hemos comentado, influye el hecho de que los procesadores mejoran sus prestaciones gracias a su mayor frecuencia, mejora de la arquitectura del núcleo y un mayor número de núcleos, mientras que las memorias se han centrado en aumentar su capacidad. En segundo lugar, los pines (situados en e chip del procesador), que permiten el acceso al bus de memoria principal o memoria DRAM (situada en módulos externos al chip del procesador), limitan el ancho de banda que se puede ofrecer entre el procesador y la memoria. En el sistema usado en este proyecto y descrito en el capítulo 4, sección 4.1, como en la mayoría de los sistemas actuales, existen 64 pines, lo que le permite un tamaño de 8 *bytes* por transferencia realizada.

Una forma adicional de abordar la pérdida de prestaciones debida a la latencia de acceso a memoria en los procesadores actuales es el mecanismo de prebúsqueda (descrita de forma más amplia en el capítulo 3, sección 3.2). Con este mecanismo se pretende sacar todo el partido a las memorias caché tratando de predecir qué datos o instrucciones van a ser usados por el procesador y llevarlos a éstas de forma especulativa. A pesar de que los *prefetchers* se empezaron a desarrollar en gran medida en los procesadores monolíticos, están jugando un papel crucial en los procesadores multinúcleo. Los procesadores multinúcleo poseen una problemática añadida ya que son múltiples núcleos los que lanzan peticiones de acceso a la memoria principal. Además, estos núcleos comparten ciertos niveles de la jerarquía de caché (normalmente la *LLC*), lo que puede introducir polución (métrica explicada en el capitulo 3, sección 3.3) en ellas. Asimismo, si disponemos de un mecanismo de *prefetch*, el ancho de banda se ve más restringido debido a los accesos bajo demanda a memoria principal a los que se debe sumar las peticiones de *prefetch* de cada núcleo. Todo ésto puede significar que se ralentice el acceso a memoria, lo que repercute negativamente en las prestaciones de las aplicaciones y del sistema en general.

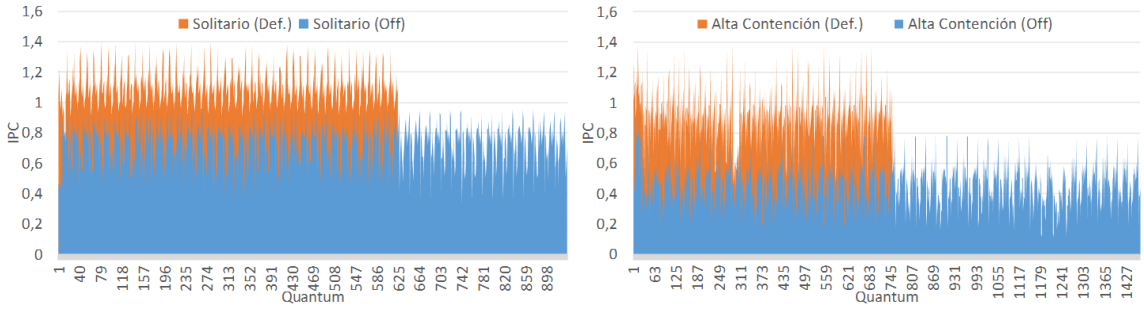
Una primera solución al problema de tener que estar compitiendo por este ancho de banda podría centrarse en eliminar o desactivar el mecanismo de *prefetch* y así disponer de mayor ancho de banda. No obstante, puesto que las prestaciones de muchas aplicaciones mejoran notablemente con a prebúsqueda agresiva, esta primera idea no es aceptable y deja abierta la resolución de la problemática de una buena gestión de este mecanismo. Muchos trabajos de investigación sobre el mecanismo de prebúsqueda en los procesadores multinúcleo, la mayoría realizados sobre simuladores, son excesivamente complejos o no tienen en cuenta ciertos parámetros de las aplicaciones en ejecución. En el capítulo 2 se describen algunas soluciones propuestas sobre este tema, las cuales precisan de hardware adicional que no se encuentra en los procesadores reales. Por otra parte, algunos procesadores recientes permiten no sólo una configuración encendido/apagado sino que permiten configurar distintos aspectos del mecanismo (por ejemplo, la agresividad), como es el caso de procesador usado en el este trabajo, **IBM POWER8** (capítulo 4, sección 4.1) con hasta 2^{25} diferentes posibles configuraciones.

1.2 Motivación

1.2.1. Motivación Profesional

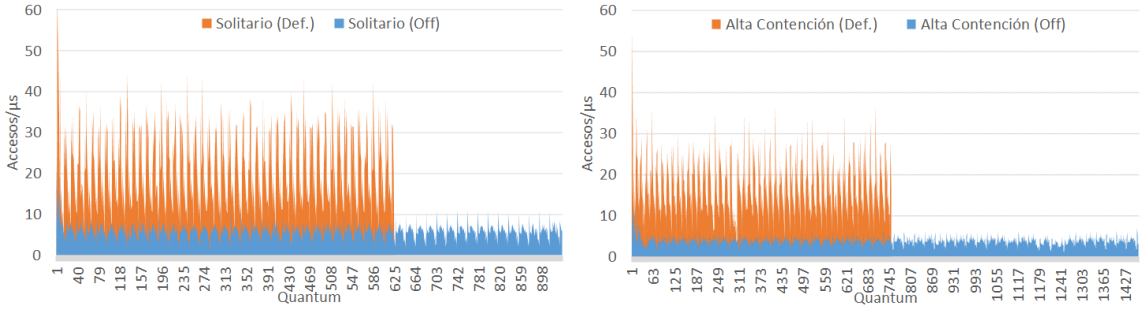
El aumento de prestaciones en los sistemas ha preocupado a los arquitectos de computadores desde el inicio de los primeros procesadores. A medida que los procesadores han evolucionado, su complejidad y prestaciones también. Actualmente existen procesadores multinúcleo, donde cada núcleo dispone de, tanto sus propios recursos, denominados privados, como compartidos. Entre los recursos compartidos podemos encontrar la memoria principal y, en la mayoría de los procesadores multinúcleo actuales, la memoria caché de último nivel de la jerarquía (*LLC*).

Los mecanismos de prebúsqueda persiguen ocultar las altas latencias de acceso a la memoria principal. Para ello utilizan recursos compartidos como el ancho de banda del bus de memoria principal y la LLC. Para mejorar esta latencia y, por ende, las prestaciones del sistema, debe conocerse el funcionamiento de la prebúsqueda y los problemas que pueden surgir debido a la interferencia entre las aplicaciones en los recursos compartidos. Esta interferencia puede afectar en gran medida a las prestaciones de las aplicaciones en ejecución concurrente.



(a) Solitario **(b)** Alta Contención

Figura 1.1: IPC de *zeusmp* en solitario y en alta contención tanto con prefetch por defecto como desactivado.



(a) Solitario **(b)** Alta Contención

Figura 1.2: Accesos/μs de *zeusmp* en solitario y en alta contención tanto con prefetch por defecto como desactivado.

Como ejemplo de esta influencia, en la Figura 1.1 se muestra la aplicación *zeusmp* tanto en una ejecución en solitario (1.1a) como en una situación de alta contención, es decir con poco ancho de banda disponible (1.1b). En el primer caso, se alcanzan unas prestaciones, cuantificadas en términos de *IPC* (instrucciones ejecutadas por ciclo del procesador) de 0,73 de media entre todas las aplicaciones con el *prefetcher* apagado y un IPC de 1,13

con el *prefetcher* por defecto. En el segundo caso, se alcanza un IPC de 0,48 con el *prefetcher* apagado y 0,98 con la configuración de del*prefetcher* por defecto.

Se puede observar en la Figura 1.2 que los accesos a la memoria principal difieren según el *prefetcher* se encuentre apagado a activado (por defecto), lo que se traduce en una variación en las prestaciones globales. Este incremento de accesos es debido a las peticiones realizadas por el *prefetcher*. Si el número de peticiones es excesivamente grande, puede repercutir negativamente en las prestaciones del sistema. Esta situación se puede observar en la ejecución de *zeusmp* con alta contención en el acceso a memoria (1.2b), donde las prestaciones bajan debido a que las interferencias que se producen en el acceso a memoria introducen una alta contención.

Las situaciones de alta contención se crean a menudo en los procesadores actuales, donde se dispone de múltiples núcleos, todos ellos compitiendo por el ancho de banda, y con el añadido del *prefetcher* de cada uno de ellos agravando todavía más el problema de la contención en el acceso a memoria. Según la tendencia tecnológica, es de esperar que el número de núcleos aumente a medida que pasen los años, por lo que este problema será cada vez más importante. Por ello, buscar una solución para evitar una excesiva (y en algunos casos, inefectiva) contención por el *prefetcher* es un tema crucial.

En este trabajo se estudian el comportamiento de las aplicaciones para finalmente escoger una configuración óptima del *prefetcher*. Con ello, se consigue una redistribución de la carga, la cual reduce la contención y las interferencias.

1.2.2. Motivación Personal

Como Ingeniero Informático estoy muy interesado en el análisis de los problemas de los computadores actuales así como la búsqueda de soluciones a los mismos.

Cuando se me presentó la oportunidad de trabajar en este tema formando parte del grupo de arquitecturas paralelas, no dudé en empezar para poder mejorar mi formación y adquirir el conocimiento adecuado para lograr entender la problemática, y así poder empezar a averiguar la causa y desarrollar una solución.

Durante el desarrollo de este proyecto hemos encontrado múltiples retos y obstáculos, como la elección de los contadores de prestaciones y entender cómo se puede modificar la configuración del *prefetcher*, los cuales hemos tenido que superar para llegar a la solución propuesta. Llegar a la propuesta presentada ha supuesto un reto y un logro personal puesto que gracias a este trabajo he adquirido mejores prácticas, paciencia y la capacidad de mantener un objetivo fijo sin perderlo de vista. Mi siguiente reto es, a partir de este punto, seguir mejorando y evolucionando la propuesta hasta tener una versión dinámica en tiempo de ejecución, y en un futuro, seguir con nuevos proyectos.

1.3 Objetivos

Los objetivos planteados en este proyecto se pueden clasificar en dos grupos principales, cada uno con sus múltiples objetivos específicos.

1. Caracterizar las aplicaciones en función de la configuración del *prefetcher* y ancho de banda disponible. El estudio persigue identificar o caracterizar diferentes tipos de aplicaciones para que nos ayude a entender su comportamiento y facilite la selección de una mejor configuración del *prefetcher*. Este objetivo global consta de los siguientes sub-objetivos.

a) Estudio del impacto de las diversas configuraciones del *prefetcher* sobre las prestaciones de las aplicaciones.

b) Análisis del consumo de ancho de banda provocado por el *prefetcher*, para alcanzar la mejora de las prestaciones (*IPC*).

c) Estudio del impacto de la contención y las interferencias en las prestaciones de las aplicaciones.

2. Desarrollo de un algoritmo de configuración del *prefetcher* en función del ancho de banda disponible. Nuestra propuesta, *BAPC*.

   Propuesta de un modelo de configuración de forma no dinámica donde se mejore el uso del mecanismo de *prefetcher* del **IBM POWER8**. El objetivo es mejorar la utilización del ancho de banda disponible. Para ello, se desactiva el *prefetcher* cuando una aplicación no se ve beneficiada, y en caso contrario, se escoge la configuración óptima. Con ello, se pretende conseguir superar las prestaciones obtenidas con la configuración por defecto. La configuración por defecto del **IBM POWER8** mejora en prestaciones notablemente a la configuración con *prefetch* desactivado. Mejorando a la configuración por defecto queda demostrada la viabilidad de una configuración adaptativa del *prefetcher*.

1.4 Impacto Esperado

La mejora a la configuración por defecto presente en el **IBM POWER8** se logra gracias a una redistribución de la carga, seleccionando la mejor configuración del *prefetcher* para cada aplicación. Esto provoca que aquellas aplicaciones que no se ven beneficiadas por el *prefetcher*, cedan ancho de banda a aquellas que sí, y además, si éstas que se ven beneficiadas muestran un compromiso entre prestaciones (*IPC*) y consumo de ancho de banda se selecciona una configuración que sea eficiente con respecto al uso del ancho de banda. Todo esto conlleva que el sistema en general logre un aumento de prestaciones para todas las aplicaciones en ejecución de entorno a un 30 % de media. En otras palabras, se incrementa la velocidad del sistema generando un ahorro temporal en la ejecución de las aplicaciones, lográndose cumplir el objetivo de ocultar de manera efectiva la latencia de los accesos a memoria principal.

1.5 Metodología

Los pasos seguidos en este trabajo para la consecución de los objetivos han sido:

1. **Estudio de propuestas existentes**. Se han estudiado las referencias bibliográficas para conocer con detalle las políticas de prebúsqueda más relevantes existentes o actualmente en desarrollo.

2. **Familiarización con el entorno de trabajo**. Solapado en el tiempo con el anterior paso, se ha ido conociendo el sistema con el que se ha realizado este trabajo. Igual que la comprensión de códigos de programa existentes con el objetivo de tener un control preciso de la máquina.

3. **Análisis del problema**. Este paso se ha desarrollado en las siguientes fases:

   a) **Caracterización de las aplicaciones en solitario**. En esta primera se ha estudiado el impacto en las prestaciones y consumo de ancho de banda de las distintas configuraciones del *prefetcher*.
   b) **Desarrollo de herramientas para la creación de escenarios**. En la segunda fase se han desarrollado herramientas con el objetivo estudiar y comparar el comportamiento de las aplicaciones bajo estudio en situaciones con un ancho de banda disponible parametrizable.
   c) **Caracterización de las aplicaciones en alta contención**. En la última fase, haciendo uso de la herramienta desarrollada en la segunda fase, se ha observado y analizado el comportamiento en un escenario de alta contención.

4. **Desarrollo de la propuesta en la máquina real**. En este paso, una vez categorizadas las aplicaciones, se ha elaborado la propuesta de configuración, eligiendo la mejor configuración del *prefetcher* en función de las características de la aplicación.

5. **Validación de la propuesta**. Último paso del proyecto, donde se ejecutan diversas pruebas con las configuraciones del *prefetcher* seleccionadas por la propuesta y se compara la mejora obtenida respecto a la misma ejecución con una configuración del *prefetcher* por defecto.

1.6 Estructura de la memoria

El resto de la memoria de este trabajo de fin de grado se organiza en los siguientes capítulos.

- Capítulo 2. Estado del arte. En este capítulo se hace un resumen de los trabajos científicos que se han publicado relacionados con temática de este trabajo. Además se destacan las principales diferencias con respecto a nuestro trabajo.
- Capítulo 3. Descripción del subsistema de memoria y los mecanismos de prebúsqueda en procesadores multinúcleo. El objetivo de este capítulo es presentar tanto al subsistema de memoria como del funcionamiento y estructura del *prefetcher*. También se explican las métricas más utilizadas para evaluar un *prefetcher*.
- Capítulo 4. Plataforma experimental. Se describen las herramientas y materiales usados para el desarrollo de este trabajo. También se presenta la metodología usada para evaluar la propuesta.
- Capítulo 5. Análisis del problema. En este apartado se realiza el estudio de las diferentes aplicaciones para evaluar su comportamiento en prestaciones *IPC* y consumo de ancho de banda ante diferentes configuraciones del *prefetcher* y ancho de banda disponible.
- Capítulo 6. Propuesta *Bandwidth-Aware Prefetch Configuration*. Se expone y explica el funcionamiento la propuesta.
- Capítulo 7. Estudio del potencial de la propuesta en cargas multiprograma, ejecución de pruebas y evaluación de los resultados.
- Capítulo 8. Conclusiones. En este capítulo se presentan las principales conclusiones obtenidas después de la realización de este trabajo así como posibles publicaciones que se derivan de él.
- Anexo A, código en *C* empleado para el lanzamiento de las mezclas.

1.7 Convenciones

En el presente trabajo se han seguido las siguientes convenciones:

- Palabras extranjeras, siglas y nombres propios de aplicaciones están escritos en *cursiva*.
- Mecanismo de prebúsqueda y prebúsqueda son usados de forma indistinta con *prefetcher* y *prefetch* respectivamente.
- Los códigos expuestos están redactados con la fuente `courier`.

CAPÍTULO 2
Estado del arte

En este capítulo se describen trabajos relacionados con el presente proyecto, tanto a nivel de investigación como a nivel académico. En este último caso, se trata sobretodo otros trabajos de fin de grado publicados en la Escuela Técnica Superior de Ingeniería Informática (ETSINF). Por último, se estudian las principales diferencias respecto a este trabajo.

2.1 Trabajos relacionados

La publicación de artículos científicos y trabajos relacionados con el tema de este trabajo son numerosos y constantes. A pesar de ello, no existe ningún trabajo en el que se evalúe el mismo tipo de cargas en un sistema real con las mismas características que el utilizado en este trabajo.

Podemos clasificar el trabajo previo en dos tipos en función del entorno en el que se realice. Estos entornos son los mecanismos de simulación y los sistemas reales. Ambos tipos de trabajos previos están centrados en variar la agresividad del *prefetcher* con el objetivo de reducir el número de accesos a memoria en un entorno multinúcleo.

La mayor parte de los trabajos se realizan sobre herramientas de simulación puesto que ofrecen una mayor flexibilidad para realizar propuestas. Por ejemplo, encontramos en [10] una propuesta llamada AC/DC, que usa una prebúsqueda adaptativa para traer datos desde memoria principal a la *LLC* (L2 en su modelo). Esta propuesta se basa en el uso de CZones [13], los cuales dividen la memoria en diversas zonas de un tamaño previamente delimitado. Además trata de encontrar patrones de acceso. El algoritmo que se propone, es adaptativo y ajusta de forma dinámica la agresividad del *prefetcher* de 2 a 16 bloques. Para aquellos casos donde se esté perjudicando las prestaciones del sistema, esta propuesta proporciona la oportunidad de desactivar el *prefetcher*. Sin embargo, no existe ninguna política para el encendido posterior.

Otra propuesta es *Feedback Directed Prefetching (FDP)* [20], se trata de una aproximación adaptativa que dinámicamente selecciona, al final de cada intervalo de muestreo, de entre 5 niveles de agresividad la que sea más conveniente. Para tomar esta decisión, utiliza métricas de precisión, retrasos e interferencia (explicadas en el capítulo 3, sección 3.3). Utiliza como *prefetcher* base un *stream prefetcher*. Esta propuesta se extendió a un entorno multinúcleo, dando lugar a HPAC (*Hierarchical Prefetcher Aggressiveness Control*). En HPAC [3] un *prefetcher FDP* se encuentra implementado en cada núcleo. Por otra parte, las decisiones se toman de forma local sobre la agresividad de cada *prefetcher*. Estas decisiones pueden ignorarse por el controlador de memoria. Éste recoge de forma global

información sobre los requisitos de memoria de cada una de las distintas aplicaciones en ejecución.

En otros trabajos como [14], se proponen técnicas de filtrado de accesos de prebúsqueda. En esta propuesta, se dispone de un filtro de mayoría ponderado para predecir la utilidad de las predicciones realizadas por el *prefetcher*. Otros trabajos similares como [24, 2] estiman a priori si el acceso que realizará el *prefetcher* será útil, en caso contrario se descartaría.

Puesto que estas propuestas son verdaderamente complejas, no se pueden implementar en procesadores comerciales. Los trabajos realizados sobre sistemas comerciales se han centrado principalmente en máquinas reales como la del presente trabajo, con el fin de obtener la mejor configuración del mecanismo de la prebúsqueda según las necesidades.

La mayoría de los procesadores comerciales no disponen de ninguna opción para controlar el *prefetcher*. En otros, como en algunos Intel, se permite activar o desactivar el *prefetcher*. Por otra parte, existen procesadores como el **IBM POWER8** utilizado en este trabajo, donde el fabricante pone a disposición una interfaz por la cual se puede cambiar esta configuración. En el capítulo 4, sección 4.1, subsección 4.1.1 se explica con detalle la forma de cambiar la configuración del *prefetcher* del **IBM POWER8**.

La propuesta que encontramos en [8], se basa en una metodología basada en *machine-learning* con el fin de seleccionar cuál de los 4 *prefetchers* disponibles en máquinas x86, Intel, se debe apagar o encender. En este tipo de máquinas, cada uno de los 4 *prefetcher* de los que dispone pueden ser apagados o encendidos de forma independiente. Esto nos lleva a un total de 16 configuraciones diferentes de *prefetcher*. La principal diferencia que podemos encontrar respecto a este estudio es que los *prefetchers* sólo se pueden apagar o encender, es decir, no existe la posibilidad de graduación.

En otra propuesta que podemos encontrar [6], el mecanismo que se propone tiene como objetivo asignar el ancho de banda de forma dinámica a cada aplicación en función de la efectividad del *prefetcher* para cada una. El fin de este mecanismo es maximizar el uso del ancho de banda de memoria. La evaluación de su propuesta se ha realizado en el **IBM POWER7**, es decir, el procesador predecesor al usado en este trabajo. Esta propuesta simplemente activa o desactiva el *prefetcher* individual de cada hilo en ejecución. Además, en el estudio de la propuesta [6] se evaluó si incluir otras configuraciones –como una agresividad intermedia– era beneficioso. La conclusión fue que la diferencia de añadir una otra configuración no era significativa. En comparación, nuestro trabajo se centra en la selección de la configuración óptima del *prefetcher* y no se limita solamente en el encendido o apagado de éste.

La metodología propuesta por PATer [7] se basa en ajustar de forma dinámica los parámetros del *prefetcher* del **IBM POWER8**. Esta propuesta hace uso de contadores de prestaciones como entrada a mecanismos de *machine-learning* para mejorar la precisión del *prefetcher*. Se utiliza un servicio en segundo plano que hace uso de los modelos de predicción para decidir cuál es la mejor configuración del *prefetcher* para las aplicaciones en ejecución. Esta propuesta sólo se centra en cargas paralelas a diferencia de nuestro trabajo, que estudia cargas multiprograma. Por último dentro de este tipo de trabajos, se encuentra libPRISM [12], que se encarga de gestionar de forma dinámica tanto el nivel *Simultaneous Multithreading (SMT)* como la configuración del *prefetcher* en el **IBM POWER8**. Sin embargo, como ocurre con PATer, se centra solamente en cargas paralelas.

2.2 Proyectos relacionados de la ETSINF

En esta sección también encontramos trabajos de los dos tipos mencionados en la Sección 2.1. Por otra parte, dentro de los trabajos de fin de grado, el tema del estudio del mecanismo del *prefetcher*, el cual se trata en este estudio, no es demasiado popular y la cantidad es reducida.

Por la parte de simulación, encontramos propuestas como la encontrada en la publicación [17], que se basa en un *prefetch* selectivo, donde *prefetchers* individuales son activados o desactivados para mejorar tanto las prestaciones como el consumo de energía de la memoria. En [17] se propone ADP, que es un *prefetcher* capaz de desactivar *prefetcher* locales en algunos núcleos cuando ellos empiezan a tener bajas prestaciones y los que se ejecutan de forma concurrente necesitan mayor ancho de banda. Cuando uno de los *prefetchers* se ha desactivado, la política para volver a activarlo se basa en una heurística la cual activa el *prefetcher* cuando predice que las prestaciones van a mejorar. Como se ha comentado, este tipo de propuesta es realmente compleja y no puede ser implementable en procesadores comerciales.

Otros como el proyecto de fin de grado [23], se basan en la evaluación de diferentes jerarquías de caché reales mediante el uso del simulador *Multi2Sim*. Hace uso de cargas multiprograma para evaluar cada una de ellas.

El estudio realizado en [1], posee ciertas similitudes con este trabajo fin de grado puesto que se realiza sobre el mismo sistema. Se trata del primer proyecto realizado sobre esta máquina donde se presenta un estudio simple de caracterización en ejecución individual – considerando *Misses Per K-Instruction (MPKI)* y el *IPC* –, y una primera tentativa de *microbenchmark*.

El presente trabajo ha conseguido grandes avances, tanto a nivel de caracterización de las aplicaciones como presentando por primera vez una propuesta de mejora. Respecto a la caracterización, se ha profundizado en otros aspectos como el ancho de banda de memoria consumido que permiten identificar mejor el comportamiento de la prebúsqueda; se ha presentando una caracterización en ejecución concurrente; y se presenta una propuesta de mejora del *prefetcher* por defecto del **IBM POWER8**.

2.3 Principales diferencias de la propuesta respecto al trabajo relacionado

Los trabajos relacionados poseen las siguientes características, las cuales son las principales diferencias respecto a la propuesta presentada en este trabajo:

- Propuestas basadas en simulación. La mayoría de ellas suelen ser muy complejas y requieren de *hardware* adicional que no se encuentra disponible en máquinas reales.
- Propuestas en máquinas con múltiples *prefetchers* simples de configuración ON/OFF. Algunos procesadores implementan múltiples *prefetchers*, que pueden activarse o desactivarse de manera selectiva, pero su funcionamiento continua siendo ON/OFF. Estos *prefetchers* no permiten controlar la actividad del *prefetcher*. En contra, el *prefetcher* utilizado en este trabajo posee 2^{25} configuraciones posibles.
- Propuestas que se centran en cargas paralelas. Estas propuestas utilizan aplicaciones compuestas por múltiples hilos, los cuales suelen tener un comportamiento similar. A diferencia de estos trabajos, en este proyecto se usan cargas multiprogra-

ma, es decir, compuestas por múltiples programas secuenciales independientes que se ejecutan concurrentemente.

CAPÍTULO 3

Descripción del subsistema de memoria y los mecanismos de prebúsqueda en procesadores multinúcleo

En este capítulo se describe el susbsistema de memoria y el mecanismo de *prefetch*, explicando su funcionamiento y características.

3.1 Subsistema de memoria en procesadores multinúcleo

Como hemos indicado anteriormente, el tiempo medio de acceso a memoria es uno de los principales aspectos que deben resolverse para mejorar las prestaciones en los procesadores actuales. El mecanismo más utilizado son las jerarquías de memorias. Los procesadores actuales suelen disponer de tres niveles de caché organizados de manera jerárquica (Figura 3.1). Cuanto más pequeñas son las memorias caché son también más rápidas. Así pues, la caché de primer nivel, integrada en el *pipeline* es la más pequeña y rápida. Asimismo, cuanto más nos alejamos del procesador y nos acercamos a la memoria principal, las memorias caché aumentan su tamaño pero con tiempos de acceso más altos.

El funcionamiento de la jerarquía de memoria es el siguiente. Cuando se ejecuta una instrucción de acceso a memoria, el subsistema de memoria empieza su búsqueda en los niveles de caché más cercanos al procesador. Si los datos se encuentran en el nivel más cercano (L1 en la Figura 3.1) se dice que es un acierto, o *hit* en terminología anglosajona. Por otra parte, si se genera un fallo (*miss*) en ese nivel de caché. Éste genera una petición del dato a un nivel inferior más alejado del procesador (si existe). En caso de que no exista un nivel más bajo, se genera un acceso a la memoria principal, lo que supone una gran latencia que repercute negativamente en las prestaciones.

El problema se agrava en los procesadores multinúcleo, ya que las peticiones de múltiples aplicaciones compiten entre ellas por el bus de memoria para acceder a la memoria principal *off-chip*. Se trata de un cuello de botella importante en los procesadores actuales, principalmente debido a que con las tecnologías actuales existe una restricción importante para poder añadir más pines al chip del procesador. En otras palabras, el número de controladores de memoria que se pueden poner en el chip del procesador se encuentra muy limitado, por lo que hay que intentar evitar ir a la memoria externa si se desean alcanzar unas buenas prestaciones.

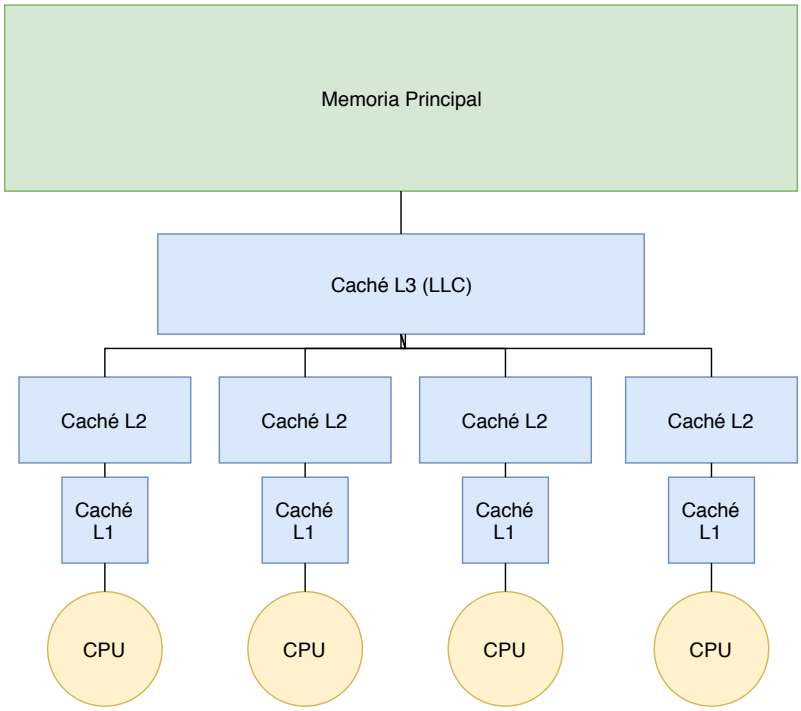


Figura 3.1: Estructura jerarquía de memoria.

Por ello, desde la introducción de los procesadores multinúcleo, los procesadores de altas prestaciones incluyen enormes memorias caché de último nivel, *LLC*. Esta caché dispone de decenas de megabytes con el objetivo de reducir los fallos de caché por falta de capacidad. Es más, los procesadores más recientes incluyen tecnologías de alta densidad para disponer de mayor capacidad de almacenamiento. Esta caché de último nivel se suele encontrar compartida entre los distintos núcleos para mejorar su utilización ya que, si un núcleo no hace uso de su espacio, otro puede utilizarlo. Esto tiene el inconveniente que puede generar interferencias entre núcleos y afectar negativamente a las prestaciones de las aplicaciones en ejecución concurrente. Por ejemplo, el **IBM POWER8** utilizado en este proyecto tiene 80MB (10 partes de 8MB) con tecnología *Embedded DRAM (eDRAM)*, y el **Intel Knights Landing** dispone de hasta 16GB con tecnología *Multi-Channel DRAM (MCDRAM)* [19].

3.2 Prebúsqueda en procesadores multinúcleo

La jerarquía de memoria es uno de los puntos cruciales en los procesadores actuales. Sin embargo, por sí misma muchas veces no es suficiente para alcanzar buenas prestaciones. Por ello, con el objetivo de sacar todo el partido a la jerarquía de memoria, se han creado una gran diversidad de mecanismos de prebúsqueda. Estos tratan de predecir qué instrucciones/datos utilizará el procesador, y trata de llevarlos cerca de él (a la/s caché/s) antes de que el procesador los solicite. Es decir, mientras la aplicación está procesando otros datos, el mecanismo de *prefetch* está accediendo a memoria y llevando los datos a caché para ocultar el tiempo de acceso. Esto supone, en el caso de acierto (*hit*), una enorme reducción en la latencia. Por otro lado, hacer una predicción exacta de los datos que va a solicitar la aplicación no es posible y las peticiones de datos pueden ser erróneas o ser lanzadas demasiado tarde o pronto.

Existen dos puntos que un buen *prefetcher* debe tener en cuenta, qué datos debe predecir el *prefetcher* y cuándo debe solicitarlos, para que los datos lleguen a tiempo [22].

Los principales inconvenientes de la prebúsqueda son que, al tratarse de un técnica especulativa, pueden haber fallos de predicción. Las predicciones erróneas, sin embargo, compiten con las peticiones bajo demanda por el consumo de recursos. Esto significa que va haber un mayor consumo de recursos, lo cual es especialmente crítico tanto en el consumo de espacio de caché como de ancho de banda de memoria. Por una parte, aparece el fenómeno conocido como polución de la caché, que se produce cuando las predicciones erróneas provocan que se reemplacen bloques útiles en la misma, con los consiguientes fallos de bloque. Además, un aumento del consumo del ancho de banda provoca que aumenten las latencias de acceso a memoria, tanto de las peticiones de prebúsqueda como de las normales, hecho que puede repercutir negativamente en las prestaciones.

Los problemas citados se agravan con los procesadores multinúcleo. Si una aplicación genera excesivas peticiones de *prefetch*, puede polucionar la caché L3 común a todos los núcleos y estar consumiendo una valiosa parte del ancho de banda que no repercute positivamente en sus prestaciones. Este ancho de banda podría ser de gran valor para incrementar las prestaciones del resto de aplicaciones.

En general, los *prefetchers* se pueden clasificar en dos grandes grupos en función de la tecnología utilizada en su implementación:

1. ***Prefetch Software***: Este tipo de *prefetcher* se basa en la generación de un tipo especial de operaciones que se añaden normalmente por el propio compilador. El objetivo de estas instrucciones es solicitar los datos que se esperan necesitar en un determinado momento futuro y llevarlos a las caché.

   Al ser necesario disponer de instrucciones específicas dentro del código, se desarrollan determinados compiladores que, mediante una serie de algoritmos, determinan la mejor posición para insertar una instrucción de *prefetch*. Estas instrucciones especiales, no bloquean el sistema en el caso que se encuentre un fallo (*miss*), sino que trabajan en paralelo a la ejecución de las instrucciones de una determinada aplicación.

2. ***Prefetch Hardware***: La mayoría de *prefetchers* de este tipo se basan en una serie de estructuras *hardware* capaces de almacenar la actividad más reciente de la memoria. Haciendo uso del *hardware* dedicado, el *prefetcher* predice qué datos solicitar y los lleva de una forma directa a una de las memorias caché. Gracias a este *hardware*, el cual no se encuentra en una ruta crítica, no contribuye a un incremento de los ciclos del procesador. Sin embargo, los mayores inconvenientes de este tipo de *prefetchers*, son la necesidad de disponer del *hardware* específico, el cual no se puede encontrar en todas las máquinas del mercado y el aumento en el consumo de ancho de banda.

Puesto que el sistema usado en este trabajo dispone de *hardware* específico para el *prefetcher*, nos centraremos en este tipo.

3.3 Métricas prefetch

Las formas de evaluar un *prefetcher* son varias. Una de las más populares es la ganancia o *speedup* de las prestaciones aunque son métricas de prestaciones del sistema. Las métricas más importantes específicas para evaluar un *prefetch* según [21] son la precisión (*accuracy*), retraso (*lateness*) e interferencia (*pollution*).

- Ganancia de prestaciones ($IPC_{speedup}$): Evalúa cuánta mejora se ha obtenido en el ámbito de las prestaciones. Su resultado indica cuánto más rápido es el sistema gracias a determinada propuesta.

$$IPC_{Speedup} = \frac{IPC\ prefetch\ activado}{IPC\ prefetch\ OFF}$$

- Precisión (*accuracy*): Mide el grado de exactitud de las predicciones del *prefetcher*.

$$Accuracy = \frac{Número\ de\ peticiones\ de\ prefetch\ válidas}{Número\ total\ de\ peticiones\ de\ prefetch}$$

- Retraso (lateness): Indica cómo de puntuales son las peticiones de *prefetch* generadas respecto a las peticiones por demanda de la aplicación. Se considera que un *prefetch* es tardío si los datos obtenidos de una petición llegan después de que se genere una petición ordinaria.

$$Lateness = \frac{Número\ de\ peticiones\ de\ prefetch\ tardías}{Número\ de\ peticiones\ de\ prefetch\ válidas}$$

- Interferencia en espacio (*pollution*): Se define como la interferencia que genera el *prefetcher* en la caché, es decir, la cantidad de bloques que remplaza que habrían sido acierto(*hit*) pero que son fallos debidos a reemplazos por prebúsquedas no útiles.

$$Pollution = \frac{Número\ de\ peticiones\ por\ demanda\ causadas\ por\ el\ prefetcher}{Número\ total\ de\ peticiones\ por\ demanda}$$

- Consumo de ancho de banda. Se trata de la relación de cuánto ancho de banda consume en relación al *prefetcher* desactivado.

$$Consumo\ de\ ancho\ de\ banda = \frac{Ancho\ de\ banda\ consumido\ por\ el\ prefetcher}{Ancho\ de\ banda\ consumido\ con\ el\ prefetcher\ desactivado}$$

- Agresividad del *prefetcher*. A diferencia de los anteriores, no se trata de una salida sino de un parámetro de configuración del mecanismo de *prefetcher*. Determina cuántos bloques de memoria se trae el *prefetcher* por cada acceso a memoria principal.

Para considerar que un *prefetcher* es efectivo, éste debe ser preciso y puntual – sin retrasos –. En otras palabras, los datos que se han llevado a caché de forma predictiva han sido correctos, y además se encontraban en la caché cuando la aplicación los ha solicitado. La dificultad radica en mejorar estas dos métricas al mismo tiempo [11]. La probabilidad de ser puntual aumenta con la agresividad, es decir, con mayor número de accesos de *prefetch* a memoria lanzados. Sin embargo, esto suele repercutir negativamente en su precisión. Además, en arquitecturas con la caché de último nivel compartida, esto implica la creación de interferencias en la *LLC*, mejorando solamente aquella aplicación que está generando la mayoría de accesos y posiblemente repercutiendo negativamente en las prestaciones del sistema en general.

CAPÍTULO 4
Plataforma experimental

En este capítulo se describen las herramientas y el *hardware* usados para el desarrollo de este trabajo.

4.1 Arquitectura del sistema

4.1.1. El procesador IBM POWER8

Este trabajo se ha desarrollado en un sistema **IBM Power System S812L** el cual incorpora un procesador **IBM POWER8**[18] con el sistema operativo Ubuntu 14.04 y el *kernel de Linux* 4.0.2. El **IBM POWER8**, puede llegar a disponer de hasta 12 núcleos funcionando hasta a 4 GHz con soporte para ejecución simultánea (*SMT*) de hasta 8 hilos por núcleo. Un diagrama de la arquitectura de este procesador se puede observar en la Figura 4.1.

La variante del procesador que se usa en este trabajo dispone únicamente de 10 núcleos funcionando a 3,69 GHz y sólo dispone de un módulo de memoria *DRAM*. Puesto que sólo se hace uso de uno de los enlaces a memoria principal disponibles, hay núcleos que se encuentran más cercanos a esta que otros.

En cuanto a la jerarquía de memoria que encontramos en el **IBM POWER8**, se compone de 3 niveles:

- *Caché de Nivel 1* (L1D y L1I) : Privada por núcleo y separada para datos e instrucciones, con una capacidad de 64KB y 32KB respectivamente por núcleo, y con tecnología *Static Random Access Memory (SRAM)*.
- *Caché de Nivel 2* (L2) : Privada por núcleo de 512KB con tecnología *SRAM*.
- *Caché de Nivel 3* (L3) : Compartida por todos los núcleos de 80MB y con tecnología *eDRAM*.

4.1.2. El prefetcher del IBM POWER8

Siguiendo con el subsistema de memoria – tema de gran importancia en este trabajo – este sistema trae consigo un sistema de prebúsqueda (*prefetcher*) muy complejo y ampliamente configurable. Para poder manejar el *prefetcher*, éste dispone de un registro de propósito específico llamado *Data Streams Control Register (DSCR)*. Este registro se compone de 12 campos, como se puede observar en la Tabla 4.1 junto a los bits asociados y la característica que controlan. La complejidad de este *prefetcher* radica en la gran cantidad

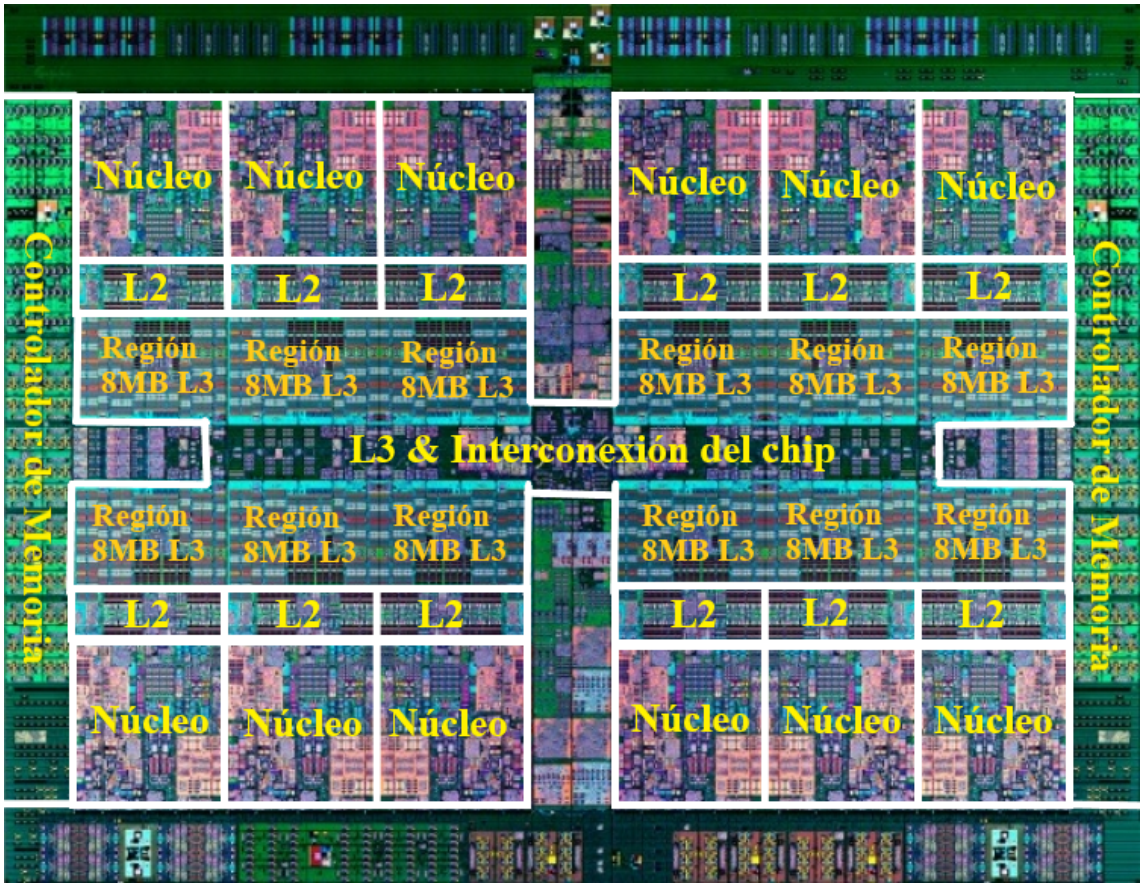


Figura 4.1: Arquitectura del procesador IBM POWER8.

de diversas configuraciones posibles, para ser exactos 2^{25}, por lo que hacer un uso óptimo de él es una ardua tarea.

Registro *DSCR*

El registro *DSCR*, se utiliza para configurar el *prefetcher*. Esta compuesto de 12 campos como se observa en la Tabla 4.1.

A continuación se explican con más detalle los campos expuestos en la Tabla 4.1:

- *Software Transient Enable* (SWTE): Se aplica el atributo de transitorio a los flujos detectados por *software*.
- *Hardware Transient Enable*(HWTE): Se aplica el atributo de transitorio a los flujos detectados por *hardware*.
- *Store Transient Enable* (STE): Se aplica el atributo de transitorio a los flujos de *stores*.
- *Load Transient Enable* (LTE): Se aplica el atributo de transitorio a los flujos de *loads*.
- *Software Unit count Enable* (SWUE): Aplica la cuenta de unidades a los flujos definidos por *software*.
- *Hardware Unit count Enable* (HWUE): Aplica la cuenta de unidades a los flujos definidos por *hardware*.
- *Unit Count* (UNITCNT): Número de unidades en un flujo de datos. Los flujos de datos que excedan este valor son finalizados.

Tabla 4.1: Estructura del registro DSCR.

bit	Nombre del campo	
0-38		
39	SWTE	Software Transient Enable
40	HWTE	Hardware Transient Enable
41	STE	Store Transient Enable
42	LTE	Load Transient Enable
43	SWUE	Software Unit count Enable
44	HWUE	Hardware Unit count Enable
45-54	UNITCNT	Unit Count
55-57	URG	Depth Attainment Urgency
58	LSD	Load Stream Disable
59	SNSE	Stride-N Stream Enable
60	SSE	Store Stream Enable
61-63	DPFD	Default Prefetch Depth

- *Depth Attainment Urgency* (URG): Representa la velocidad o prioridad con la que las ráfagas de prebúsquedas llegan a la profundidad fijada. De manera equivalente al *DPFD*, el valor 0 selecciona la urgencia por defecto (urgencia 4). Así pues, la urgencia propiamente dicha varía desde 1 (sin urgencia) hasta 7 (máxima urgencia).
- *Load Stream Disable* (LSD): Este campo configura la detección e inicialización de ráfagas de *loads* (*load streams*).
- *Stride-N Stream Enable* (SNSE): Activa la detección *hardware* e inicialización de flujos de *loads* y *stores* que tienen una separación (*stride*) mayor que un solo bloque de caché. Estos flujos de *loads* pueden ser detectados cuando el campo *LSD* es 0, y los flujos de *stores* cuando el campo *SSE* se encuentra a 0 [4].
- *Store Stream Enable* (SSE): Activa la detección *hardware* e inicialización de flujos de *stores*.
- *Default Prefetch Depth* (DPFD): Representa la profundidad de la prebúsqueda en número de bloques. Como se ha comentado anteriormente, un valor 0 selecciona la profundidad por defecto (4 bloques). Por otro lado, el valor 1 indica profundidad nula o prebúsqueda desactivada (independientemente de otros campos de la configuración). Por lo tanto, la profundidad en número de bloques puede ser configurada entre los valores 2 ($DPFD = 010_2$) y 7 ($DPFD = 111_2$).

Dentro de las configuraciones posibles hay algunos casos especiales, como por ejemplo, cuando en este registro se encuentran todos los campos a 0, es decir *DSCR*=0, se encuentra en su configuración por defecto. Esta configuración equivale a que los campos de profundidad (*DPFD*) y urgencia (*URG*), se encuentren en un valor equivalente a 4 para cada uno de ellos. Además, no sólo estos dos campos se encuentran activos, sino que existe un campo con lógica negativa (*LSD*), que como almacena un 0 significa que la característica *Load Stream* se encuentra activa. En esta configuración el resto de campos quedan desactivados.

Este trabajo se centra en estudiar la posible viabilidad de una configuración adaptativa del *prefetcher*, la cual mejore las prestaciones del sistema respecto a la configuración por defecto. Por esta razón, se debe estudiar el comportamiento del sistema respecto a las diferentes configuraciones posibles. Debido al gran número de posibles configuraciones, en este trabajo se ha optado por simplificar el análisis y se ha centrado en aquellos

campos que podrían ser más influyentes, la urgencia (*URG*) y la profundidad (*DPFD*). El resto de campos se mantendrán en su configuración por defecto.

Para la caracterización de las aplicaciones, tanto en solitario como en situaciones de alta contención, modificamos estos campos en sus valores máximos y mínimos a la vez para valorar su influencia en las prestaciones y en el ancho de banda. En la Tabla 4.2 encontramos las configuraciones usadas.

Tabla 4.2: Configuraciones usadas.

Configuración	Especificación	Valor DSCR
U1P2	URG=1 DPFD=2	66
U1P7	URG=1 DPFD=7	71
U7P2	URG=7 DPFD=2	450
U7P7	URG=7 DPFD=7	455

4.1.3. Configuración del prefetcher

Este registro se puede modificar tanto para el sistema en general o para un identificador de proceso (*Process Identifier (PID)*); en este trabajo se accede a él de dos formas distintas:

1. Mediante un comando de *bash* proporcionado por *ppc64_cpu*:

```
# Cambio DSCR para el sistema en general
ppc64_cpu --dscr=$VALOR
# Cambio DSCR para un PID
ppc64_cpu --dscr=$VALOR -p $PID
```

2. Mediante código en *C* (se muestra un programa que permite cambiar el valor dado un identificador de programa (*PID*)):

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <unistd.h>
#include <string.h>
#include <sys/ptrace.h>
#include <errno.h>

#define PTRACE_DSCR 44

static int do_dscr_pid(int dscr_state, pid_t pid)
{
  int rc;
    rc = ptrace(PTRACE_ATTACH, pid, NULL, NULL);
  if (rc) {
    fprintf(stderr, "Could not attach to process %d to %s the "
      "DSCR value\n%s\n", pid, (dscr_state ? "set" : "get"),
      strerror(errno));
    return rc;
  }
    wait(NULL);

  rc = ptrace(PTRACE_POKEUSER, pid, PTRACE_DSCR << 3, dscr_state);
  if (rc) {
    fprintf(stderr, "Could not set the DSCR value for pid "
      "%d\n%s\n", pid, strerror(errno));
```

```
28      ptrace(PTRACE_DETACH, pid, NULL, NULL);
29      return rc;
30    }
31
32      rc = ptrace(PTRACE_PEEKUSER, pid, PTRACE_DSCR << 3, NULL);
33    if (errno) {
34      fprintf(stderr, "Could not get the DSCR value for pid "
35        "%d\n%s\n", pid, strerror(errno));
36      rc = -1;
37    } else {
38      printf("DSCR for pid %d is %d\n", pid, rc);
39    }
40
41    ptrace(PTRACE_DETACH, pid, NULL, NULL);
42    return rc;
43  }
44  int main(int argc, char *argv[])
45  {
46    pid_t pid = getpid(); //PID del que se tiene que cambiar
47    int dscr_val; //Valor del dscr
48    printf("Ejecutando...\nTengo el pid: %d.\n",pid);
49    printf("Introduce el pid objetivo: ");
50    scanf("%d",&pid);
51    printf("Introduce el valor de DSRC: ");
52    scanf("%d",&dscr_val);
53
54    do_dscr_pid(dscr_val, pid);
55      sleep(1);
56      printf("Fin...\n");
57  }
```

Este código requiere el uso de la función *ptrace*, que permite cambiar el valor del registro DCSR para cada proceso. Esta función, descrita de la siguiente forma,

$$long\ \ ptrace(enum\ \ \_\_ptrace\_request\ \ request,\ pid\_t\ pid,\ void\ *addr,\ void\ *data);$$

permite que un proceso (observador, *tracer*), pueda supervisar y controlar la ejecución de otro proceso (observado, *tracee*). Esto permite cambiar y examinar la memoria y registros usados por el observado.

Puesto que para hacer uso de la función explicada hay que asociar los procesos, la primera parte del código (l. 14-21) usa la solicitud *PTRACE_ATTACH* para empezar a controlar al proceso cuyo identificador es *pid*. Una vez asociado, la segunda parte (l. 24-30), usa la solicitud *PTRACE_POKEUSER*, la cual permite acceder a un desplazamiento dentro de la pila del proceso observado y cambiar el valor por otro pasado por parámetro. La tercera parte (l. 32-39), se usa como comprobación de que el valor se ha escrito de forma correcta mediante el uso de la función *PTRACE_PEEKUSER*, que permite leer una parte de la pila de memoria del proceso observado. La última parte (l. 41), hace uso de *PTRACE_DETACH* para liberar el proceso observado y así desvincularlo, ya que de no hacerlo podría provocar un error en el proceso.

El resto del código (l. 44-57) genera unas preguntas por consola para que de forma sencilla e intuitiva se cambie el valor del registro en un proceso en concreto.

4.2 Contadores de prestaciones

La evaluación de las prestaciones puede suponer una tarea compleja además de importante en los sistemas actuales, puesto que su capacidad de computo y rendimiento se ha ido incrementando junto con la complejidad para su configuración. Por esta razón, monitorizar el comportamiento del sistema es un tema clave para poder observar las prestaciones que se están obteniendo, y así, poder evaluar la efectividad de una configuración o mejora añadida.

Existen varias formas de obtener información sobre las prestaciones o más métricas. La primera sería mediante instrucciones en el propio código fuente u opciones en el compilador, lo que aumentaría y modificaría el código que se quiere evaluar tanto en tamaño como en complejidad. La segunda forma sería mediante contadores de prestaciones, los cuales acceden a un *hardware* específico dentro del procesador para leer contadores de prestaciones. Gracias a estos contadores *hardware*, no es necesario modificar el código que queremos evaluar. Estos monitores de prestaciones son más conocidos como *Performance Monitoring Unit (PMU)* que hace uso de unos registros de propósito específico independientes del resto del procesador pensados como contadores de prestaciones *hardware*. Estos monitores dependen por completo de la arquitectura del sistema, por lo que se debe prestar especial atención a los contadores disponibles en el sistema que se quiere evaluar. En concreto, el **IBM POWER8** dispone de seis registros para contadores de prestaciones, donde cuatro de ellos pueden configurarse para monitorizar un evento diferente; los otros dos son fijos y miden las instrucciones ejecutadas y los ciclos usados. Estos contadores pueden medir ciclos, instrucciones, fallos de caché, accesos a memoria y así hasta 1144 eventos distintos.

Para poder medir estos contadores existen diversas herramientas, en particular en este trabajo se ha hecho uso de la herramienta *perf*[1], que forma parte del *kernel de Linux* desde 2009 (v. 2.6.31) a través de la librería *libpfm*[2]. Ésta es una librería que ayuda a configurar los contadores proporcionando una interfaz genérica dada una lista de contadores de prestaciones. Además, dispone de traducciones de los nombres de los contadores a los nombres en crudo. Los contadores usados en este estudio forman parte del proyecto de software libre que ofrece un perfil para estadística, este proyecto es *Oprofile*[3].

La lista de eventos monitorizados –entre paréntesis el nombre en formato *Oprofile*– en este estudio son los siguientes:

- *cycles (PM_CYC)*: Número de ciclos usados en la ejecución de la aplicación.
- *instructions (PM_INST_CMPL)*: Número de instrucciones ejecutadas por la aplicación.
- *LLC-Load-Misses (PM_DATA_FROM_L3MISS)*: Número de fallos de *loads* en la *LLC* en la ejecución de la aplicación. Equivalente al número de accesos a de memoria principal sin contar los realizados por el *prefetcher*.
- *PM_Mem_Pref*: Número de accesos a la memoria principal realizados por el *prefetcher* en la ejecución de la aplicación.

[1] http://man7.org/linux/man-pages/man1/perf.1.html
[2] http://perfmon2.sourceforge.net/
[3] http://oprofile.sourceforge.net/docs/ppc64-power8-events.php

Una vez obtenidos los contadores mencionados podemos derivar ciertas métricas que realmente nos son de utilidad a la hora de evaluar la ejecución de una o varias aplicaciones. Para este estudio, las métricas usadas han sido:

- *Instruciones por ciclo (IPC)*: Número de instrucciones ejecutadas por ciclo de la aplicación.

$$IPC = \frac{Instrucciones\ ejecutadas\ (instructions)}{Ciclos\ usados\ (cycles)}$$

- *Ancho de banda (accesos/µs)*: Número de accesos a memoria principal realizados por μs por la aplicación en su ejecución.

$$Accesos/\mu s = \frac{(LLCLoadMisses + PM\_Mem\_Pref) * 3690}{cycles}$$

  El 3690 que se encuentra en la operación es para poder convertir de ciclos de procesador funcionando a 3,69 GHz a μsegundos.

- *Aceleración (speedup)*: Usado para poder identificar el porcentaje de mejora de una configuración a otra.

$$Speedup = \frac{IPC\ de\ la\ configuracion\ a\ evaluar}{IPC\ de\ la\ configuracion\ base}$$

4.3 Benchmarks SPEC CPU 2006

Este estudio se ha realizado mediante los *benchmarks* de la suite de *Standard Performance Evaluation Corporation* **SPEC CPU 2006**[4] con las entradas *reference*. Éstas sirven para obtener una medida comparable o estándar con el resto de computadores sobre nuevas propuestas o cambios. Con el objetivo de evaluar con el mismo peso a todas estas aplicaciones, se han ejecutado de forma individual con el *prefetcher* desactivado durante 120 segundos; una vez finalizada esta ejecución, se ha medido el número de instrucciones ejecutadas y se ha almacenado para la realización de todos los demás experimentos. Además también se han usado para formar mezclas, con lo que así obtenemos las cargas multiprograma, hecho que resulta útil para evaluar la propuesta realizada en este trabajo.

A continuación se hace un breve comentario sobre el tipo de carga de cada una de las aplicaciones [5].

4.3.1. Benchmarks de aritmética entera

bzip2 (401.bzip2): Escrito en **C**, esta versión del programa está basada en la versión de Julian Seward. La única diferencia entre ambos es que la versión aquí utilizada no realiza ninguna lectura o escritura de ficheros excepto el de entrada. Tanto la compresión como la descompresión está realizada completamente en memoria, lo que permite aislar el trabajo realizado al procesador y el subsistema de memoria.

mcf (429.mcf): Escrito en *C*, este programa está derivado de *MCF*, un programa usado para la planificación de vehículos en el transporte público masivo. Considera sólo un destino y vehículos homogéneos.

[4] https://www.spec.org/cpu2006/

hmmer (456.hmmer): Escrito en *C*, hace uso de los modelos ocultos de Markov para una la búsqueda de patrones en secuencias de ADN. Esta técnica se usa para hacer un uso intensivo de búsqueda en base de datos.

sjeng (458.sjeng): Escrito en *C*, está basado en *Sjeng* 11.2, el cual es un programa que juega al ajedrez y diversas variantes de él. Intenta encontrar el mejor movimiento por una búsqueda en árbol. La versión *SPEC*, es una versión mejorada modificada para que refleje la carga de programas profesionales actuales.

libquantum (462.libquantum): Escrito en *C99*, *libquantum* es una librería para la simulación de un procesador cuántico. Implementa el algoritmo descubierto por Peter Shor para factorización de números, aportando una estructura para representar un registro cuántico y diversas puertas lógicas elementales. Realiza múltiples operaciones de factorización.

h264ref (464.h264ref): Escrito en *C*, es una implementación de H.264/AVC (Advanced Video Coding). Este programa codifica vídeo utilizando diversos parámetros como buena compresión y codificación rápida, o mejor compresión.

omnetpp (471.omnetpp): Escrito en *C++*, es una simulación de una gran red *Ethernet*, está basado en el sistema de simulación discreta *OMNeT++*.

astar (473.astar): Escrito en *C++*, está derivada de la librería de enrutamiento 2D, la cual es usada en la inteligencia artificial de los vídeo juegos.

xalanxbmk (483.xalancbmk): Escrito en *C++*, es una versión modificada de *Xalan-C++*, un procesador *XSLT* escrito con un subconjunto de *C++*. Este programa se encarga de transformar documentos de tipo *XML* en otros tipos como *HTML*.

4.3.2. Benchmarks de aritmética en coma flotante

bwaves (410.bwaves): Escrito en *Fortran*, es una simulación numérica de ondas de choque en flujos viscosos laminares, transónicos y tridimensionales. Este programa se encarga de resolver las ecuaciones de Navier-Strokes usando el algoritmo *Bi-CGstab*, el cual resuelve sistemas de ecuaciones lineales no simétricas de forma iterativa.

gamess (416.gamess): Escrito en *Fortran*, realiza un amplio rango de cálculos cuánticos químicos.

milc (433.milc): Escrito en *C*, utiliza una versión secuencial del programa su3imp, el cual es a la vez importante y relevante porque las prestaciones paralelas dependen de las prestaciones de un solo procesador.

zeusmp (434.zeusmp): Escrito en *Fortran*, es una versión basada en *ZEUS-MP*, un código de cálculo de dinámicas de fluido. El problema físico resuelto en la versión de **SPEC CPU2006** es la simulación de un choque de ondas en tres dimensiones con presencia de un campo magnético uniforme a lo largo del eje x.

gromacs (435.gromacs): Escrito en *C* y *Fortran*, es una versión derivada de GROMACS, un paquete versátil que realiza dinámicas moleculares. Esta versión, realiza una simulación de la proteína *Lysozyme* en una solución de agua e iones.

cactusADM (436.cactusADM): Escrito en *Fortran* y *C*, es una combinación de *Cactus*, un entorno de resolución de problemas de código libre, y *BenchAMD*, un núcleo computacional representativo de muchas aplicaciones de relatividad numérica. Este programa resuelve las ecuaciones de evolución de Einstein, las cuales describen las curvas espacio-temporales como respuesta a su contenido de materia.

leslie3d (437.leslie3d): Escrito en *Fortran*, está derivado de *LESlie3d* (Large Eddy Simulations ith Linear-Eddy Model in 3D), un código para el cálculo de dinámicas de flui-

do a nivel de investigación usado para investigar un amplio abanico de fenómenos de turbulencia.

namd (444.namd): Escrito en *C++*, está derivado de NAMD, un programa paralelo para la simulación de grandes sistemas biomoleculares. Casi todo su tiempo de computo se emplea en el cálculo de interacciones interatómicas.

soplex (450.soplex): Escrito en *C++*, está basado en *SoPlex*, que resuelve un problema lineal usando el algoritmo *Simplex*.

povray (453.povray): Escrito en *C++*, es una técnica de renderizado que calcula una imagen de una escena por una simulación de cómo los haces de luz viajan en el mundo real pero de forma inversa, es decir, a diferencia de la vida real donde el haz de luz es emitido por la fuente y rebota en los objetos y este reflejo es captado por el ojo humano o el sensor de la cámara. En este algoritmo, los rayos son emitidos por la cámara y cada impacto debe ser calculado.

GemsFDTD (459.GemsFDTD): Escrito en *Fortran*, este programa resuelve las ecuaciones de Maxwell en tres dimensiones en el dominio del tiempo, haciendo uso de diferencias finitas. (*finite-diference time-domine* (FDTD)).

4.4 Herramientas

Además de lo mencionado en este mismo capítulo, se han utilizado diversas herramientas para la realización de este trabajo. Éstas son las siguientes:

- *Shell Scripts*: Los archivos de este tipo son secuencias de comandos que se lanzan uno detrás de otro de principio a fin. Es muy útil cuando se quiere automatizar el lanzamiento de una gran cantidad de comandos quitando la necesidad de tener que hacerlo de forma manual.

```
1 for dscr in 0 1 64 67 450 455; do
2  {./20180108_executa_instruccions_120_selectCore -A 6 -B 16 -N 1 -d
       200;} 2>> resultados.txt; &
3  pid1=$!
4  frase=‘ppc64_cpu --dscr=$dscr -p $pid1‘
5  echo $frase
6  wait $pid1
7 done
```

- *Programas en C*: En el caso de este trabajo, son los programas que se encargan de lanzar los *benchmarks*, así como medir los contadores de prestaciones mediante la librería *libpfm* –mencionada en este mismo capítulo– y controlar que su ejecución dure el número de instrucciones correspondiente. Para la ejecución de mezclas se usa un código que también se encarga de controlar el valor de la configuración del *prefetcher* para cada una de las aplicaciones que la componen. Se ha partido de la infraestructura *software* base desarrollada por Josué Feliu durante su tesis doctoral. Este *software* se ha modificado en gran medida para hacer posible el acceso a diferentes contadores de medida así como posibilitar el acceso a los registros de configuración de los *prefetchers* individuales de cada aplicación y hacer posible la implementación de la propuesta. La versión de código utilizada para el lanzamiento de la propuesta está disponible en el Anexo A. En este código se hace uso de la librería *libpfm* (capítulo 4, sección 4.2) además de la función de cambio de *DSCR* por código explicada en el capítulo 4, sección 4.1, subsección 4.1.2. Como utilizamos un núcleo por aplicación, se hace uso de las estructuras y funciones que dispone Linux

para poder llegar a este fin. La función que se debe usar para asignar un núcleo es *sched_setaffinity* y su interfaz es la siguiente:

$$int\ sched\_setaffinity\ (\ pid\_t\ pid\ ,\ unsigned\ int\ cpusetsize\ ,\ cpu\_set\_t\ *mask\ )$$

Los argumentos son los siguientes:

1. ***PID***: Identificador de proceso al que queremos asignar la máscara.
2. *cpusetsize*: Cómo de grande es la máscara.
3. *mask*: Puntero a la máscara. El tipo de este argumento es *cpu_set_t*, que para facilitar el manejo de este tipo de variable, en este trabajo se hace uso de $CPU\_ZERO()$ y CPU_SET. La primera función elimina los núcleos asociados a un proceso, y la segunda añade el núcleo que queremos asignar a la máscara.

Los resultados obtenidos de la ejecución, por lo general estaban formateados de la siguiente forma:

```
En general:
Counters: cycles intructions LLC-Load LLC-Load-Misses PM_Mem_Pref
Ejemplo:
Counters: 451138584248 777305661802 41962560 181875 3334

En mezclas:
Counters: benchmark cycles intructions LLC-Load LLC-Load-Misses
    PM_Mem_Pref
Ejemplo:
Counters: bzip2 464154673113 558352458674 1151242600 24454008 4386424
```

- *Comandos Bash*: Instrucciones que se han usado directamente en el terminal de Linux para una función puntual y específica. Se ha usado en la gran mayoría de las veces para la ejecución de funciones con el objetivo de obtener una mejor presentación de los datos y métricas. Por ejemplo:

```
cat resultados.txt | grep "Counters:" | awk '{print $2 \
"\t" $3 "\t" $4 "\t" $5 "\t" $6 "\t" $3/$2 \
"\t"  (($5+$6)*3690)/$2}' >> resultadosFinal.txt

Entrada:
 Counters: 453189753043 558772139597  877988125 3564202 60611136
Resultado:
 453189753043 558772139597  877988125 3564202 60611136  1.23298
     0.522534
```

- *Microsoft Excel*[5]: Su uso se ha limitado al almacenamiento de los resultados y su tratamiento para la creación de las gráficas necesarias.

[5]https://products.office.com/es-es/excel

CAPÍTULO 5
Análisis del problema

5.1 Caracterización de las aplicaciones en solitario

En esta sección se estudia y caracteriza la forma en la que la configuración del *prefetcher* afecta a las prestaciones de las aplicaciones (SPEC CPU2006) en ejecución individual. Para lograrlo, se han escogido ciertos campos de la configuración del *prefetcher* como son la urgencia (*URG*) y la profundidad (*DPFD*) en sus valores máximos y mínimos –puesto que disponen de 3 bits–, además de estos valores, también se han estudiado las configuraciones por defecto y no *prefetch* (prebúsqueda desactivada). Concretamente, las configuraciones utilizadas han sido *OFF* (*DPFD*=1, no *prefetch*), *DEF* (*DSCR*=0, por defecto), *U1P2*, *U1P7*, *U7P2* y *U7P7*, donde *UxPy* corresponde a urgencia (*URG*) *x* y profundidad (*DPFD*) *y*. Como se especifica en el capítulo 4, sección 4.1, subsección 4.1.2 , el valor mínimo de la profundidad es 2, a diferencia de la urgencia que es 1.

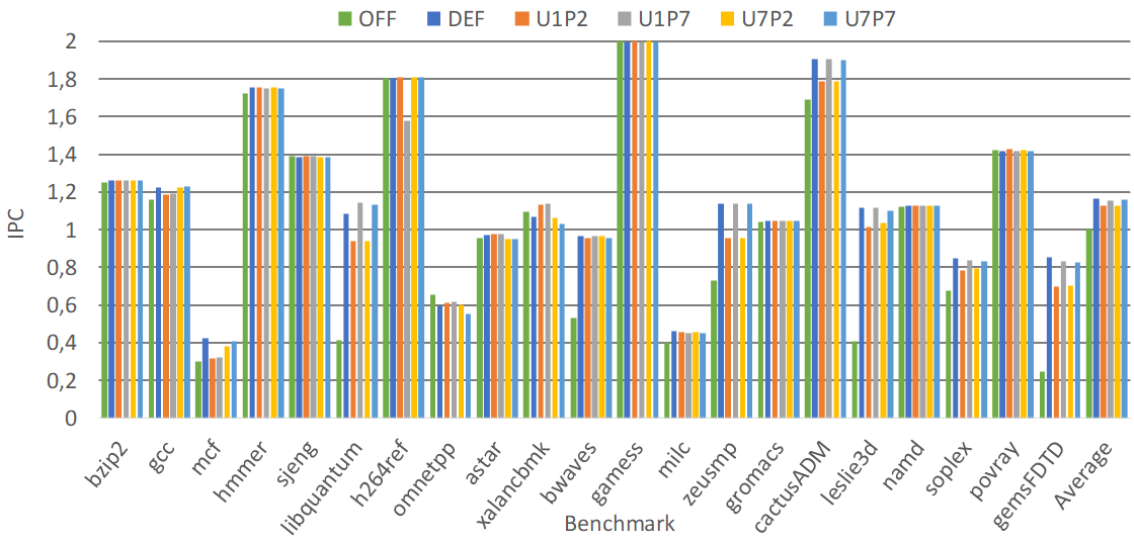


Figura 5.1: *IPC* de las aplicaciones en ejecución individual.

La Figura 5.1, muestra las prestaciones (*IPC*) de cada aplicación con cada una de las configuraciones del *prefetcher* estudiadas. Analizando estos resultados, se pueden realizar tres observaciones importantes:

- Se puede observar como a pesar que las prestaciones de algunas aplicaciones no mejoran al utilizar el *prefetcher*, otras muchas incrementan sus prestaciones de una

manera muy significativa. Por ejemplo, podemos ver como *libquantum*, *zeusmp*, *leslie3d* o *gemsFDTD* llegan a mejoras de hasta mas del 100 %. Es por tanto que para estas aplicaciones es crucial que el *prefetcher* se mantenga siempre activo.

- Se aprecia cómo, en general, el campo de la profundidad tiene mayor relevancia en las prestaciones que la urgencia. Esta situación se observa claramente en *libquantum* y *zeusmp*. Sin embargo, en otras aplicaciones como *gcc* y *mcf*, la urgencia tiene una mayor influencia en las prestaciones.

- Superar las prestaciones obtenidas con una configuración del *prefetcher* por defecto es una tarea complicada, aunque en algunas configuraciones con una alta profundidad (DPFD) llegan a igualarse.

El ancho de banda de memoria principal es un recurso crítico en los procesadores multinúcleo actuales (sobretodo durante la ejecución de cargas multiprograma) y muy utilizado por el *prefetcher*. La caracterización también estudia el ancho de banda consumido por cada configuración del *prefetcher*. La métrica usada para estudiar el consumo de ancho de banda de memoria principal ha sido los accesos por microsegundo.

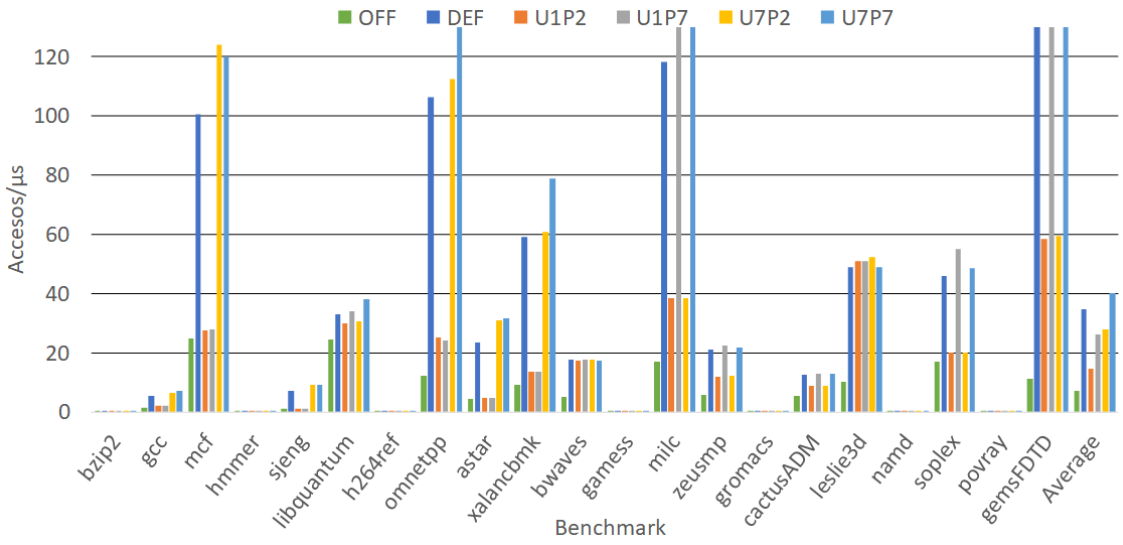


Figura 5.2: Accesos/μs consumidos por las aplicaciones en ejecución individual.

De forma similar a la Figura 5.1, la Figura 5.2 muestra las mismas aplicaciones y configuraciones del *prefetcher* pero, en este caso, se han mostrado los accesos por μs consumidos en cada configuración. Como se puede observar, queda patente la variación del ancho de banda de una configuración de la prebúsqueda a otra, incluso para aquellas aplicaciones en las que las prestaciones no se ven sensiblemente afectadas (por ejemplo, *milc*). Por otra parte, en otras aplicaciones como *xalanc*, la escasa mejora obtenida en las prestaciones se consigue incrementando considerablemente el consumo de ancho de banda. Como última apreciación, el campo que más influencia tiene en el consumo de ancho de banda es el de profundidad (*DPFD*).

5.2 Desarrollo de un microbenchmark

Los resultados obtenidos en la ejecución en solitario representan el comportamiento de la aplicación cuando ésta dispone de todo el ancho de banda de memoria principal para ella sola. Con el objetivo de estudiar el impacto del ancho de banda disponible en las prestaciones en estas aplicaciones, se ha diseñado e implementado un *microbenchmark*. Entendemos por *microbenchmark* a una aplicación que nos permite controlar el uso de un recurso en concreto, de una forma precisa y configurable. Éste nos permite estudiar las prestaciones de un sistema bajo ciertos escenarios previamente descritos, en relación al ancho de banda disponible en memoria principal.
Esencialmente, el *microbenchmark* realiza accesos de forma continua a la memoria principal con una tasa de accesos configurable. Esto restringe esta interferencia únicamente a la memoria principal, lo que nos permite que el estudio sobre el impacto en las prestaciones sea más riguroso.

Algoritmo 5.1: Pseudocódigo del *microbenchmark*

```
1: int A[ARRAY_SIZE]
2: while true do
3:    for (i = 0; i < ARRAY_SIZE; i=i+stride) do
4:       A[i]++
5:    end for
6:    for (i = 0; i < #nops; i++) do
7:       asm("nop")
8:    end for
9: end while
```

En el Algoritmo 5.1 presenta el pseudocódigo del funcionamiento del *microbenchmark*. Éste posee dos fases. La primera de ellas (líneas 3-5), realiza los accesos a memoria. La segunda (líneas 6-8), se trata de una espera activa mediante la inserción de instrucciones *nop*. El número de *nops* es configurable y permite controlar del ancho de banda que consume el *microbenchmark*.

La primera de las dos fases se encarga de realizar los accesos a un *array* de tamaño mayor que la caché de último nivel (*LLC*). Los accesos consecutivos se separan mediante una distancia, llamada *stride*.

El desarrollo de esta aplicación se ha divido también en dos etapas. Una primera donde se ha implementado el *microbenchmark*, y una segunda en la que se ha graduado el control de su comportamiento en el acceso a memoria principal. En esta segunda fase, se han tenido que ajustar dos parámetros, el *stride* y el número de instrucciones *nops* insertadas.

En la primera fase, dedicada a la implementación del *microbenchmark*, se ha usado como base el *software* proporcionado por Josué Feliu [16]. Éste se ha modificado para llegar a los requerimientos deseados. El código final es el siguiente:

```
1
2 #include <sched.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <sys/time.h>
6
7 int main (int argc, char *argv[]) {
```

```
8   int i, j, k, mem_accesses = 0;
9   int *A;
10  int nop_ops, iter, acces, core;
11  int size = 1024*1024*40; //160MB
12  if (argc != 5)
13  {
14    printf("Error (Numero de argumentos = %d). Uso prg1 iter nop_ops stride
          core\n", argc);
15    return 1;
16  }
17  A = (int *) malloc (sizeof (int *) * size);
18  sleep (5);
19  iter = atoi (argv[1]);
20  nop_ops = atoi(argv[2]);
21  acces = atoi(argv[3]);
22
23  while (1)
24  {
25    for (k=0; k<size; k+=acces)
26    {
27      A[k]++;
28      mem_accesses += 1;
29    }
30
31    for (i=0; i<nop_ops; i++)
32    {
33      asm("nop");
34    }
35
36  }
37  sleep(5);
38  return 0;
39 }
```

En el código expuesto podemos observar lo siguiente:

- **Línea 17**. Reserva de espacio en memoria del tamaño del *array*.
- **Línea 18**. Espera de 5 segundos antes de empezar los accesos a memoria.
- **Líneas 23-26**. Bucle infinito principal. Mediante este bucle se consigue que el programa funcione de manera continua.
- **Líneas 25-29**. Bucle de acceso a memoria. Se lee la variable y se incrementa. La variable *mem_accesses* sirve para guardar el número de accesos realizados desde el código.
- **Líneas 31-33**. Bucle de espera activa. Lanza el número de instrucciones *nop* pasadas como parámetro.

La segunda fase, la graduación para el control del *microbenchmark*, se ha iniciado con el control del *stride*. Este es la distancia entre dos accesos consecutivos al vector reservado en memoria. Si la distancia entre estos dos accesos es demasiado pequeña, la prebúsqueda lo detectara y el comportamiento podría no ser el esperado. Por otra parte, si es grande, la prebúsqueda será más compleja. Como podemos observar en la Figura 5.3, se ha ido aumentando el *stride* hasta observar que el consumo se ha estabilizado. En este estudio hemos concluido que con un *stride* de 128 elementos del *array*– 512 bytes– este consumo se mantiene constante.
Por otra parte, para controlar los accesos realizados a memoria principal, se ha ajustado

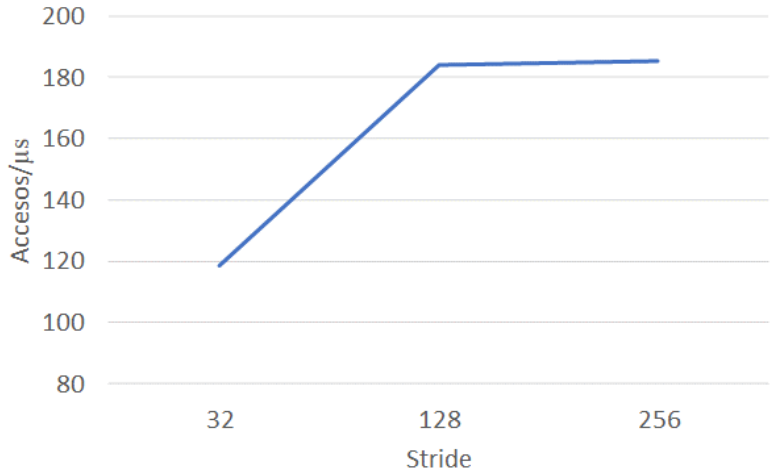


Figura 5.3: Accesos/μs consumidos por el *microbenchmark* en función del *stride* .

el número de instrucciones *nops* insertadas de modo que podamos variar del 100 % de accesos por μs hasta un 10 % en intervalos de 10 % sobre el total. De este modo, como se puede apreciar en la Figura 5.4, el número de *nops* puede variar desde 0 *nops*, para un máximo consumo, hasta 10 millones de *nops*, para consumir una décima parte del total. Es decir, cuando aumentamos el parámetro #*nops*, se reduce el consumo y por consiguiente la contención en el acceso a memoria principal.

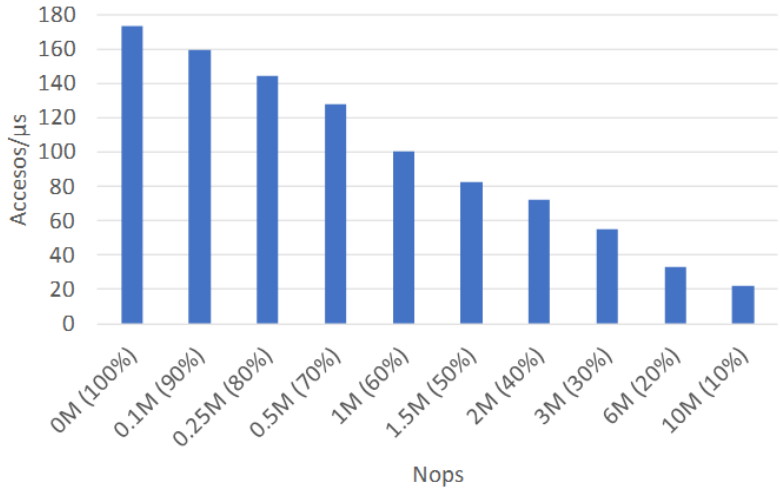


Figura 5.4: Accesos/μs consumidos en función del número de *nops* en el *microbenchmark*.

Para aumentar aún más el consumo del *microbenchmark* se han ejecutado varias instancias del *microbenchmark*. Como se observa en la Figura 5.5, el número de accesos por μs aumenta ligeramente cuando se ejecuta más de una instancia. Del mismo modo, se aprecia que de 3 a 4 instancias este aumento no es nada significativo, por lo que para los experimentos donde se hace uso del *microbenchmark* se han usado 3 instancias con el fin de alcanzar la máxima contención. Los resultados obtenidos demuestran que se llegan a alcanzar los 189 accesos/μs, siendo éste el máximo soportado por el sistema.

Como conclusión, podemos afirmar que gracias al diseño de este *microbenchmark* disponemos de una herramienta útil para controlar el ancho de banda disponible para la aplicación que queremos evaluar. De esta forma, podemos analizar el comportamiento de las aplicaciones bajo diferentes niveles de contención de memoria.

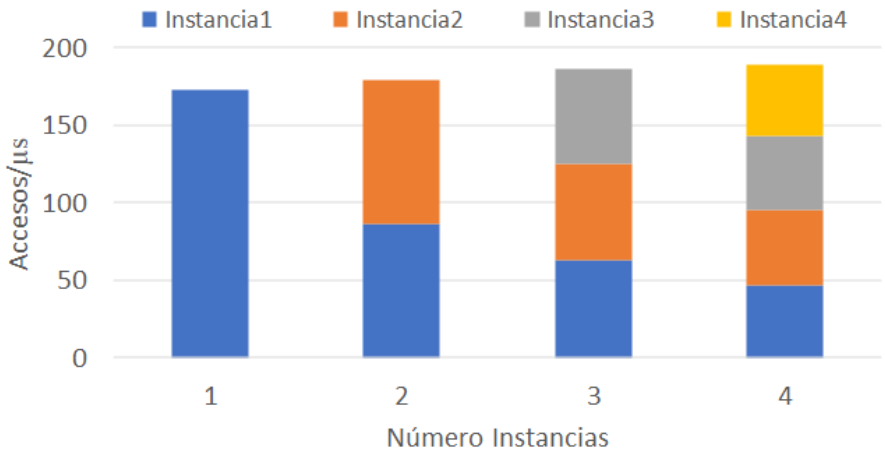


Figura 5.5: Accesos/μs consumidos respecto el número de instancias del *microbenchmark*.

5.3 Caracterización de las aplicaciones junto el microbenchmark

Esta sección analiza como la contención en el ancho de banda de memoria afecta a las prestaciones las aplicaciones, tanto en prestaciones (*IPC*) como en consumo de ancho de banda. Cada una de las aplicaciones se ha ejecutado de forma concurrente junto con tres instancias del *microbenchmark*, con el objetivo de crear congestión en la memoria principal y así limitar el ancho de banda disponible para la aplicación. Para evitar las posibles interferencias a otros recursos que no sea la memoria principal, la aplicación a evaluar y cada una de las instancias del *microbenchmark* se han ejecutado en núcleos distintos.

5.3.1. Caracterización con interferencia máxima

Para obtener la mayor interferencia en el ancho de banda posible, cada instancia del *microbenchmark* se ha ejecutado con el máximo consumo posible (#$nops = 0$). De manera análoga a la sección 5.1, se han utilizado las mismas aplicaciones y configuraciones del *prefetcher* para evaluar cada aplicación. La Figura 5.6 muestra el *IPC* alcanzado por cada aplicación cuando se encuentra con un ancho de banda limitado. La figura muestra que, como ocurría en su ejecución en solitario, la desactivación de la prebúsqueda tiene efectos negativos en las prestaciones de algunas de las aplicaciones. A diferencia de la ejecución individual, se observa que el campo de la configuración del *prefetcher* que mayor influencia tiene en las prestaciones es la urgencia. Esto se debe a que cuando existe contención en el acceso a memoria, la prioridad que adquieren las peticiones–en otras palabras, la velocidad con la que se sirven– es más importante que la agresividad de éstas.

Como ocurrió en la ejecución en solitario, las prestaciones alcanzadas por las aplicaciones están asociadas a un incremento del ancho de banda como podemos observar en la Figura 5.7. Ésta figura representa el consumo de ancho de banda de la aplicación (segmentos de color nítido), y de forma apilada (segmentos difuminados) el consumido por las instancias del *microbenchmark*. El total de ambas dos es prácticamente idéntico al ancho de banda máximo soportado por el sistema. Como hemos comentado, a diferencia de lo que podemos observar en ejecución individual (Figura 5.2), la urgencia tiene mayor peso que la profundidad.

Con ambos resultados en mano, podemos encontrar, entre las aplicaciones que mejoran de una forma significativa sus prestaciones cuando la prebúsqueda se encuentra activa (es decir, *prefetch friendly*), dos grupos diferenciados desde el punto de vista de prestaciones y consumo de ancho de banda.

- Aplicaciones *prefetch-confguration sensitive*: Aquellas sensibles a la configuración del *prefetch* desde un punto de vista de prestaciones (por ejemplo, *gcc,mcf,zeusmp,cactus*). Se puede observar como el *IPC* de algunas aplicaciones como *zeusmp* puede variar significativamente, en este caso un 30 %, de una configuración a otra (U1P2 es un 30 % peor que U7P2).
- Aplicaciones *prefetch-configuration insensitive*: Aquellas que, desde un punto de vista de prestaciones, se mantienen prácticamente estáticas de una configuración a otra (por ejemplo, *hmmer, sjeng, h264ref*) o éste oscila muy ligeramente (por ejemplo, *bwaves* o *libquantum*). Es decir, con el *prefetcher* siempre activo, son insensibles al valor de la urgencia o profundidad. Por otra parte, si tenemos en cuenta el consumo de ancho de banda entre configuraciones, a pesar de tener un *IPC* similar éste varía enormemente hasta en un factor superior a 10×.

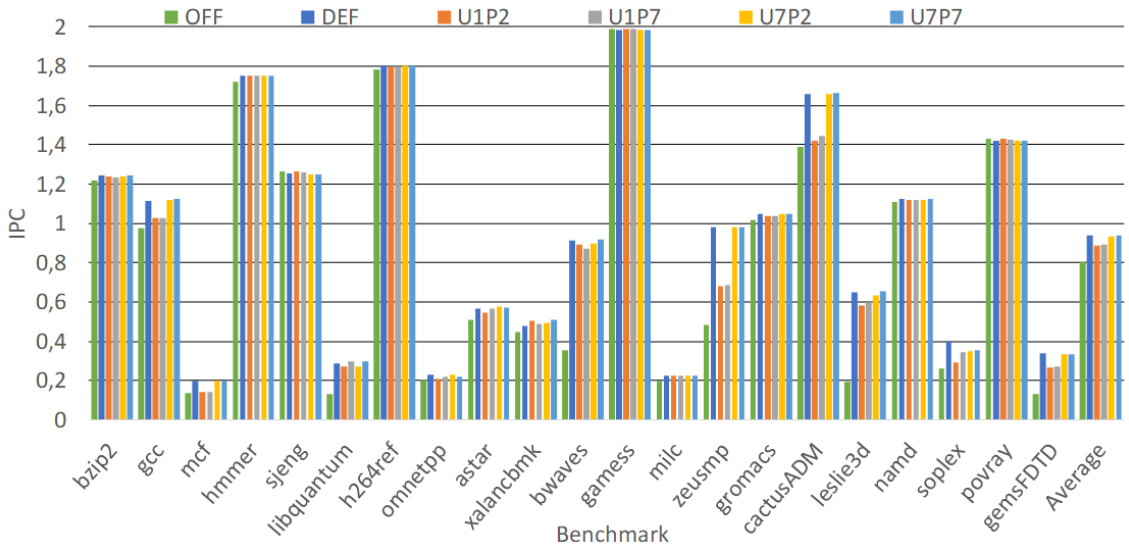


Figura 5.6: Prestaciones para cada aplicación ejecutándose con tres instancias del *microbenchmark* configuradas con el máximo consumo de ancho de banda.

Dado que el ancho de banda es un recurso limitado, con esta caracterización en mente, una buena política de configuración del *prefetcher* debe tener en cuenta las limitaciones de ancho de banda presentes, y seleccionar las configuraciones que requieran de un menor consumo de ancho de banda para aplicaciones del tipo *prefetch-configuration insensitive*. Así mismo, deberá buscar una relación de compromiso entre incremento de prestaciones y consumo de ancho de banda para las del tipo *prefetch-configuration sensitive*.

5.4 Sensibilidad del IPC al ancho de banda disponible

En la sección anterior se ha estudiado el impacto en las prestaciones cuando las aplicaciones se encuentran en condiciones de alta congestión de ancho de banda puesto que

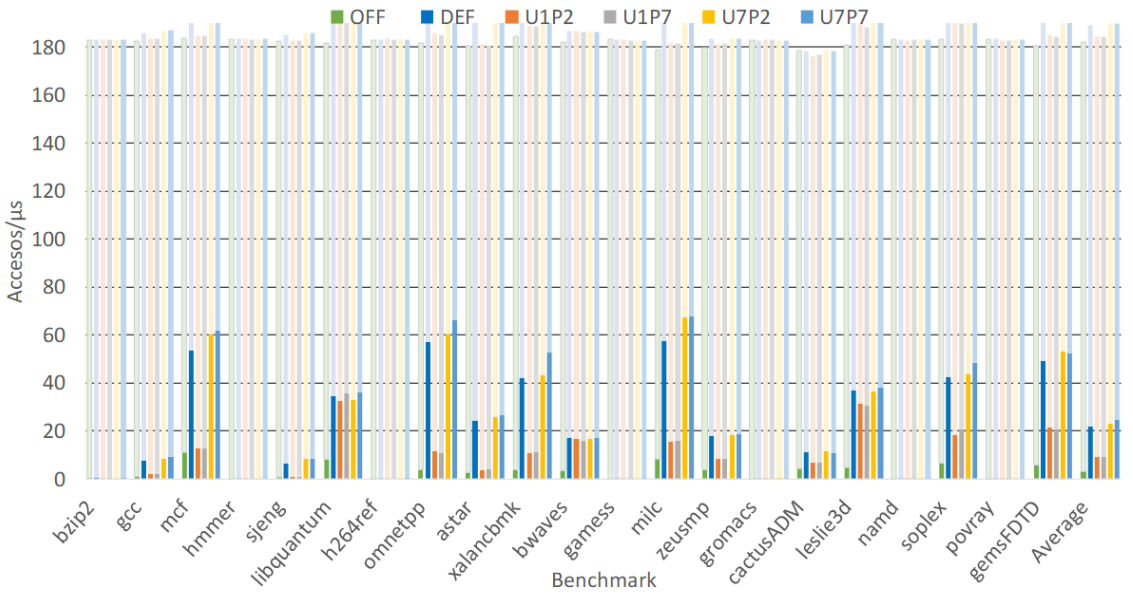


Figura 5.7: Accesos/μs para cada aplicación ejecutándose con tres instancias del *microbenchmark* configuradas con el máximo consumo de ancho de banda.

se ejecutaban junto con tres instancias del *microbenchmark*, lo que llegaba al consumo total del ancho de banda de la memoria principal. Por otro lado, en esta sección se estudia el impacto que tiene el ancho de banda disponible sobre las prestaciones obtenidas en las aplicaciones. Como se dispone de un *microbenchmark* parametrizable, podemos variar su consumo de ancho de banda de un 100 % (0 *#nops*) hasta un 0 % (equivalente a ejecución en solitario), por lo que podemos observar el comportamiento de la aplicación a estudiar cuando el ancho de banda disponible se va incrementando.

En la Figura 5.8 encontramos la sensibilidad al ancho de banda para los *benchmarks* de tipo entero como un ejemplo de este comportamiento. Para la obtención de esta gráfica tanto las aplicaciones a estudio como el *microbenchmark* se han mantenido con una configuración del *prefetcher* por defecto (*DSCR*=0).

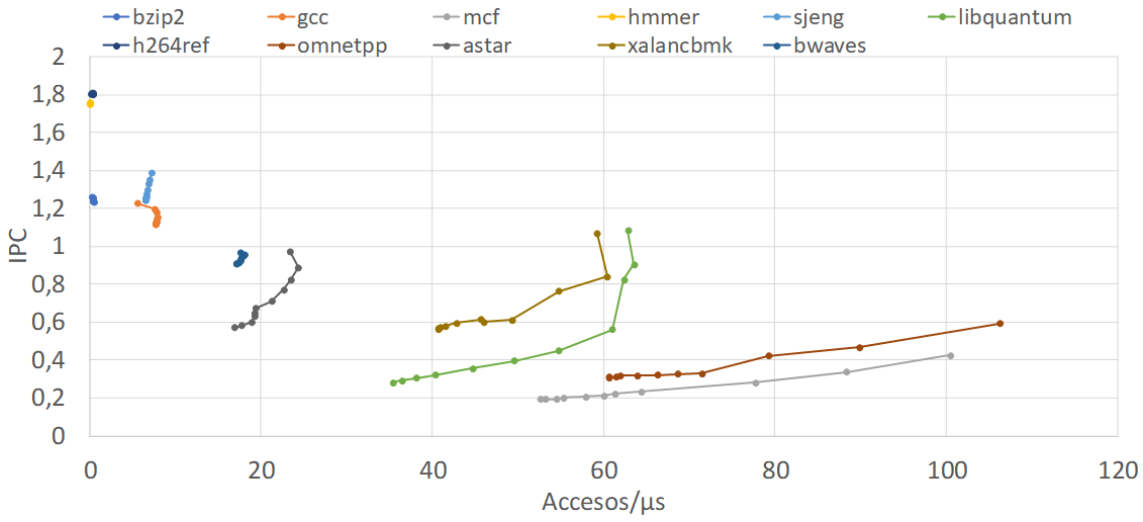


Figura 5.8: *IPC* respecto el consumo de ancho de banda para los *benchmarks* enteros. *Prefetch* por defecto tanto en la aplicación como en el *microbenchmark*.

Gracias a la Figura 5.8 podemos identificar varios tipos de aplicaciones:

- Aplicaciones con muy pocos accesos a memoria las cuales, como era de esperar, son insensibles al ancho de banda de memoria como es *hmmer*.
- Aplicaciones con pocos accesos (4-8 accesos/μs): Con un mínimo incremento de ancho de banda, su *IPC* varía. Por ejemplo, el *IPC* de *sjeng* varía de un *IPC* en solitario de 1,4 a uno de 1,2 cuando se encuentra en compañía de las instancias del *microbenchmark*. Esto es debido a que el número de accesos por μs decrementa un 11,7 % ya que pasa de 7,15 accesos/μs en su ejecución en solitario a 6,4 en condiciones de alta contención. La situación de contención impide que las peticiones sean servidas con mayor rapidez, lo que implica que sus prestaciones se vean afectadas negativamente.
- Aplicaciones con más de 30 accesos/μs, necesitan un gran incremento de ancho de banda para lograr una mejora de prestaciones (por ejemplo, *mcf* o *libquantum*).

Nótese que las aplicaciones de este último grupo requieren de una política de configuración del *prefetcher* tal que permita un consumo de ancho de banda suficiente para que obtengan unas prestaciones adecuadas.

5.5 Conclusiones

Una vez entendido el comportamiento de las aplicaciones, podemos concluir que es necesaria una buena política de configuración del *prefetch*. Ésta debe tener en cuenta no sólo aquellas aplicaciones que no se benefician de él —y desactivarlo— sino que, además, para aquellas que sí lo hacen, controlar que si existe mejora en prestaciones no se debe a un excesivo aumento de consumo de ancho de banda. En caso de no controlarlo, el consumo excesivo de ancho de banda repercutiría negativamente en las prestaciones de las demás. De esta forma, gracias a una buena gestión del *prefetcher*, las prestaciones del sistema en general se verán beneficiadas.

CAPÍTULO 6
Propuesta: Bandwidth-Aware Prefetcher Configuration

En este capítulo se presenta una configuración del *prefetcher* para procesadores multinúcleo. Se trata de la primera propuesta para cargas multiprograma.

6.1 Diseño de la propuesta

La propuesta presentada recibe el nombre de *Bandwidth-Aware Prefetcher Configuration* (*BAPC*). Como se ha estudiado en el Capítulo 5, desde el punto de vista de prestaciones en ejecución individual, superar las prestaciones obtenidas con una configuración por defecto es muy complicado puesto que funciona muy bien en esas situaciones. Por otro lado, en situaciones de alta contención, es decir, cuando el ancho de banda de memoria es limitado, como sucede en los procesadores multinúcleo, elegir la mejor configuración para cada aplicación es una tarea complicada, ya que la elección de una configuración repercute en el resto de aplicaciones ejecutadas concurrentemente, y por tanto, en las prestaciones globales del sistema.
BAPC se basa en seleccionar y asignar una configuración del *prefetch* determinada a cada aplicación en función de su propio comportamiento y del resto de aplicaciones que se ejecutan de forma concurrente junto a ella. El algoritmo propuesto tiene en cuenta las características obtenidas de las diversas categorías de aplicaciones identificadas en el estudio de caracterización. Como se ha estudiado, el *IPC* de las aplicaciones clasificadas como *prefetch-configuration sensitive* se ve afectado negativamente cuando hay una deficiencia de ancho de banda disponible.

Si una política de configuración del *prefetcher* no tuviera en cuenta el ancho de banda consumido por las aplicaciones provocaría que las aplicaciones clasificadas como *prefetch-configuration insensitive* malgastaran ancho de banda disponible, ya que ésta no aprovecharía los datos que la prebúsqueda le aportaría, puesto que son insensibles a la configuración elegida. El ancho de banda extra consumido por las peticiones de *prefetch* puede afectar a otras aplicaciones clasificadas como *prefetch-configuration sensitive* que se puedan ejecutar de manera concurrente. Por lo tanto, el objetivo de la propuesta es incrementar el ancho de banda disponible para aplicaciones del tipo *prefetch-configuration sensitive* con tal de seleccionar la mejor configuración desde el punto de vista de prestaciones. Para lograr esto, la configuración del *prefetcher* para aplicaciones del tipo *prefetch unfriendly* y *prefetch-configuration insensitive* debe ser la que menor ancho de banda consuma. Nótese que esta configuración no afecta las prestaciones de éstas aplicaciones y, sin embargo, sí ayuda a mejorar las prestaciones de las aplicaciones en ejecución concurrente.

Algoritmo 6.1: *BAPC*

1: Identificar aplicaciones:
2: → *prefetch unfriendly*
3: → *prefetch-configuration sensitive*
4: → *prefetch-configuration insensitive*
5: Para las apl. *prefetch unfriendly*:
6: → desactivar el prefetcher
7: Para las apl. *prefetch-configuration insensitive*:
8: → elegir la configuración del prefetcher con menor consumo de ancho de banda.
9: Para las apl. *prefetch-configuration sensitive*:
10: → descartar configuraciones con prestaciones a costa de elevado consumo de BW
11: → elegir configuración con mejores prestaciones

El algoritmo presentado tiene 2 puntos principales. La primera parte (líneas de la 1 a la 4) consiste en identificar el tipo de aplicación, según la taxonomía presentada en el capítulo 5. La segunda parte (líneas de la 5 a la 11) consiste en aplicar criterios distintos según la categoría de la aplicación. La selección de la mejor configuración se ha basado en los siguientes criterios, los cuales se resumen en el Algoritmo 6.1.

- Aplicaciones *prefetch unfriendly*: Son aquellas cuyas prestaciones no obtienen ningún beneficio por el uso de la prebúsqueda, por lo que para este tipo de aplicaciones, mantener el *prefetcher* desactivado es lo más conveniente para lograr tener un ancho de banda mayor para el resto de las aplicaciones.
- Aplicaciones *prefetch-configuration insensitive*: Son aquellas aplicaciones que se benefician de la prebúsqueda pero sus prestaciones no varían de una configuración a otra. Por tanto, se debe seleccionar aquella configuración que haga un menor consumo de ancho de banda con el mismo objetivo que para las *prefetch unfriendly*.
- Aplicaciones *prefetch-configuration sensitive*: Son aquellas aplicaciones que sus prestaciones sí que varían de una configuración del *prefetcher* a otra; por ello, debe cuantificarse esta mejora de prestaciones respecto al consumo de ancho de banda. Si para lograr un ligero incremento de prestaciones se requiere un gran incremento de consumo de ancho de banda, esta configuración queda descartada y se selecciona una configuración que, a pesar de tener prestaciones ligeramente menores, su consumo de ancho de banda sea menor. El criterio de selección de mejor configuración considera la relación entre el incremento de las prestaciones y el coste en términos de ancho de banda. Para ello, se ha definido la métrica aceleración por ancho de banda o $speedup/\Delta_{BW}$ que cuantifica el ratio entre el *speedup* y el incremento del ancho de banda consumido, ambos respecto a la configuración de *prefetch* desactivado.

$$S/\Delta_{BW} = \frac{Speedup\ sobre\ OFF}{\Delta_{BW}\ sobre\ OFF}$$

  En el presente trabajo, la selección de la configuración se ha basado en aquella que maximice la métrica mencionada siempre que sea mayor que 0,25. Este número significa que la mejora del *IPC* debe ser al menos un 25 % del incremento del ancho de banda consumido por la aplicación. En caso contrario, se considera que el exceso de ancho de banda consumido no compensa la ganancia del IPC y, por tanto, se escoge la configuración con *prefetch* desactivado. Por ejemplo, si una aplicación consigue mejorar las prestaciones (*IPC*) en un factor de 1,10 a costa de un incremento del

consumo de ancho de banda cuantificado en un factor de 2, entonces esta configuración es elegible, pero en el caso que este incremento de consumo de ancho de banda suponga un factor de 5, ésta dejaría de ser elegible y se optaría por desactivar la prebúsqueda.

CAPÍTULO 7
Estudio del potencial de la propuesta en cargas multiprograma

En este capítulo se analiza el potencial de la propuesta en sistemas multinúcleo con cargas multiprograma.

7.1 Diseño de cargas multiprograma

Con el fin de estudiar los beneficios en prestaciones de la propuesta *BAPC* en procesadores multinúcleo, en este trabajo se han diseñado varias mezclas compuestas por múltiples aplicaciones secuenciales para crear situaciones de alto consumo de ancho de banda de memoria principal. Para elegir las aplicaciones que formarán parte de estas mezclas, sólo se han considerado elegibles aquellas que en ejecución individual realizan un consumo superior a los 5 accesos/μs en nuestro sistema.

Tabla 7.1: Aplicaciones estudiadas y configuración del *prefetcher*.

Benchmark	**Configuración**
mcf	DEF
omnetpp	OFF
astar	OFF
xalancbmk	U1P2
milc	U1P2
zeusmp	DEF
cactusADM	DEF
soplex	DEF
gemsFDTD	DEF

En la primera columna de la tabla 7.1, se muestran nueve aplicaciones que cumplen esta condición. De este conjunto, se han seleccionado de forma aleatoria aplicaciones para la creación de cargas multiprograma con el fin de evaluar la propuesta. En concreto, se han generado cuatro cargas, dos de ellas están formadas por ocho aplicaciones, y las otras dos, por diez.
Aplicando los criterios descritos en la sección 6.1 de manera no dinámica, se han identificado las configuraciones seleccionadas por *BAPC* (aplicando el algoritmo descrito en el capítulo 6) para cada una de las aplicaciones. La configuración seleccionada para cada

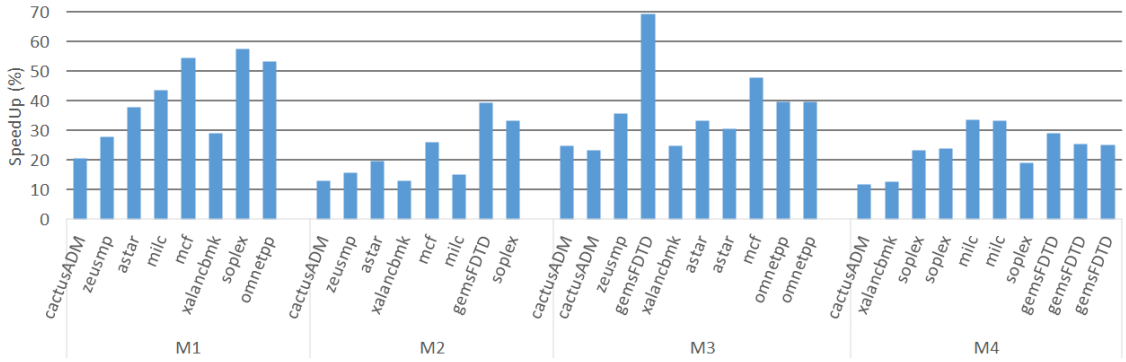


Figura 7.1: *Speedup* de cada una de las aplicaciones de las 4 mezclas con la configuración seleccionada por *BAPC* con respecto a la configuración por defecto.

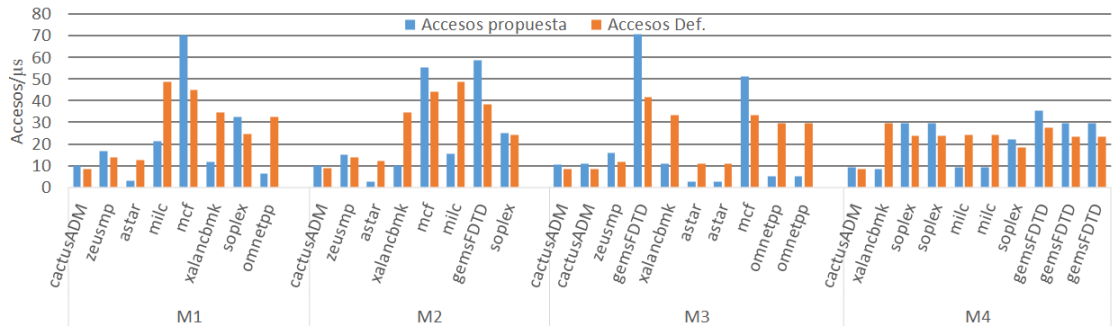


Figura 7.2: Ancho de banda consumido por cada aplicación de las 4 mezclas estudiadas.

aplicación se muestra en la segunda columna de la Tabla 7.1.

La Figura 7.1 muestra el *speedup* por aplicación para cada una de las 4 mezclas alcanzado por *BAPC* respecto a la configuración *prefetch* por defecto que mostró tan buenos resultados en ejecución individual. Como se puede apreciar, los resultados obtenidos son de gran interés puesto que se llegan a mejoras de casi un 70 %, siendo la media de la mejora de un 30 %.

Las mejoras alcanzadas se consiguen gracias a dos aspectos. Por una parte, la distribución de los accesos a la memoria principal, como se puede observar en la Figura 7.2, donde la distribución de la carga de una configuración a otra es totalmente diferente. Por otra parte, el número total de accesos por microsegundo es menor, lo que consigue reducir la contención, tal y como se muestra en la figura 7.3. En consecuencia, se consigue una mejora en todas las aplicaciones, incluso en aquellas a las que se reduce el ancho de banda respecto a la configuración por defecto. Gracias a ello, se alcanza una mejora general en todo el sistema, independientemente de su sensibilidad al uso de la prebúsqueda.

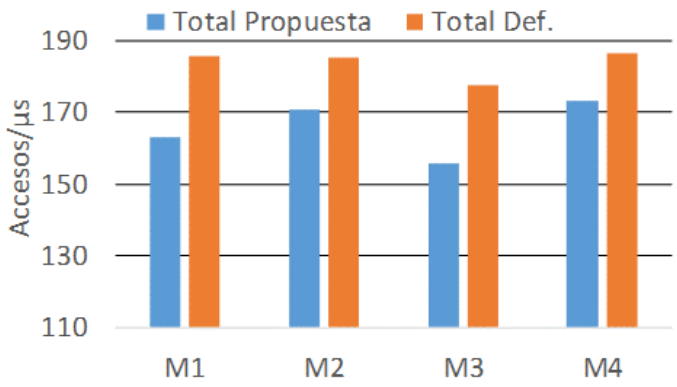


Figura 7.3: Ancho de banda consumido por cada mezcla tanto por *BAPC* como por la configuración por defecto.

CAPÍTULO 8
Conclusiones

En este último capítulo se presentan las principales conclusiones de este trabajo, así como la relación existente con el grado cursado, publicaciones derivadas y trabajos futuros.

8.1 Visión general

Con el objetivo de ocultar la latencia de acceso a memoria, en los procesadores actuales se implementan una serie de *prefetchers*, los cuales se encargan de solicitar y llevar los datos –de forma especulativa– al procesador antes de que éstos se necesiten. En procesadores como el usado en el presente trabajo, el **IBM POWER8**, se presentan múltiples *prefetchers* avanzados que permiten al usuario modificar la configuración de su comportamiento para cada aplicación de una forma dinámica. Por otro lado, puesto que el número de configuraciones posibles es muy amplio y esto influye en cómo afecta la prebúsqueda en las aplicaciones, esta tarea resulta muy compleja.

En este trabajo se ha caracterizado la sensibilidad tanto a nivel de prestaciones (*IPC*) como de ancho de banda consumido por las aplicaciones de la suite SPEC CPU2006, para una serie de configuraciones posibles del *prefetcher* del **IBM POWER8**. El estudio presentado en este trabajo ha identificado diversos tipos de aplicaciones desde el punto de vista de prestaciones. Entre aquellas que se ven beneficiadas por el *prefetcher*, *prefetch friendly*, podemos encontrar aquellas cuyo comportamiento se ve muy afectado (*prefetch-configuration sensitive*) y otras que se muestran completamente insensibles (*prefetch-configuration insensitive*) a las variaciones de la configuración. Esta observación, junto con el estudio del consumo de ancho de banda presentado, nos deja con una importante valoración, puesto que cada configuración implica un consumo diferente de ancho de banda, el cual influye en gran medida en las prestaciones globales del sistema.

Basándose en esta observación, en este trabajo se ha propuesto *Bandwidth-Aware Prefetcher Configuration* (*BAPC*). Se trata de un algoritmo de configuración del *prefetcher* para procesadores multinúcleo con cargas multiprograma que tiene en cuenta no sólo las prestaciones, sino también el ancho de banda de memoria. Para analizar el potencial de la propuesta, se ha usado la información del estudio de caracterización para seleccionar la mejor configuración del *prefetcher* individual para cada aplicación y, se han generado de manera aleatoria distintas mezclas, las cuales se han lanzado tanto con una configuración por defecto como con la configuración seleccionada por *BAPC*.

Los resultados obtenidos muestran que la propuesta *BAPC* consigue una mejora en prestaciones de hasta un 70 % en comparación a la configuración por defecto en algunas aplicaciones y, de media, la mejora se sitúa entorno a un 30 %.

8.2 Relación del trabajo desarrollado con los estudios cursados

En primer lugar, asignaturas como Estructura de Computadores (ETC), Arquitectura e Ingeniería de Computadores (AIC) y Arquitecturas Avanzadas (AAV) han sido necesarias para disponer de los conocimientos necesarios sobre arquitecturas, y más en concreto, el subsitema de memoria y su funcionamiento. Además, en AAV se ha estudiado una introducción a los contadores de prestaciones, necesario para la evaluación del sistema.

Para la realización de los experimentos, el lanzamiento de ellos y el tratamiento de los datos, se han usado lenguajes aprendidos y desarrollados durante el grado como son *C* o *ShellScript*. Además de un amplio conocimiento del sistema operativo *Linux* y el funcionamiento de su terminal (*bash*)– aportado por asignaturas como Fundamentos de Sistemas Operativos (FSO), Diseño de Sistemas Operativos (DSO) o Seguridad en los Sistemas Informáticos (SSI) –.

Gracias a la realización de memorias para las partes prácticas de la mayoría de las asignaturas cursadas, se ha podido mejorar tanto la expresión escrita como la organización de la información; lo que ha sido de gran ayuda para la redacción de este trabajo.

En cuanto al apartado de competencias transversales, encontramos las siguientes:

- **Aplicación y pensamiento práctico y Aprendizaje permanente**: Han sido necesarios los conocimientos ya aprendidos y el aprendizaje de nuevos, como el uso del sistema de este trabajo y el acceso a los contadores, y su posterior representación. Todo ello para lograr los objetivos esperados y analizar las razones de las mejoras.
- **Análisis y resolución de problemas y Conocimiento de problemas contemporáneos**: El problema tratado en este trabajo está presente en todos los sistemas informáticos, por ello, identificar y analizar las causas es esencial. En este trabajo se ha realizado un análisis del por qué no se llegan a las prestaciones que se debería. A partir de esto, en este trabajo se ha realizado la propuesta con el fin de mejorar el sistema en general.
- **Instrumental específica**: Como se ha nombrado a lo largo del proyecto, ha sido necesario el uso de un procesador específico, el **IBM POWER8** por su increíble capacidad de configuración del *prefetcher*. Además de esto, herramientas propias como código en *C* o el uso del sistema *Linux* como base del desarrollo, han sido esenciales para la realización de los experimentos y con ello, este trabajo.

8.3 Publicaciones derivadas y trabajos futuros

Este trabajo ha derivado en una publicación la cual se presentará en las **XXIX Jornadas de Paralelismo**[1] en Septiembre del 2018.

Se pretende mejorar la propuesta en este trabajo, en concreto, diseñar una versión dinámica en tiempo de ejecución de *BAPC*. Es decir, la idea es que la propuesta identifique el tipo de las aplicaciones aplicando el algoritmo en tiempo de ejecución, sin necesidad de realizar un análisis off-line del comportamiento de las aplicaciones. Esperamos conseguir

[1] http://www.jornadassarteco.org/descripcion/?anyo=2018&simposio=jp2018

la propuesta dinámica en un plazo de dos o tres meses para su envío a una conferencia internacional de primer nivel.

También se pretende estudiar el comportamiento de la propuesta con otras cargas, como pueden ser la última versión de las SPEC CPU (SPEC CPU2017) [2], o cargas paralelas.

Por último mencionar que los beneficios alcanzados por la propuesta son conservativos puesto que son una primera aproximación, y se requieren refinamientos sucesivos. Estos son susceptibles de mejorar mediante un estudio más exhaustivo de los 12 campos restantes del registro *DSCR* para la configuración del *prefetcher*.

También cabe la posibilidad de adaptar este trabajo a otras máquinas reales a pesar de no tener un *prefetcher* con las mismas características. Con esto se afirmaría si la mejora obtenida es particular de la máquina o puede ser generalizada, lo que sería un logro importante.

[2] https://www.spec.org/cpu2017/

Bibliografía

[1] Jahel Carmona Vila. Evaluacion experimental de los mecanismos de prebusqueda en el ibm power8, 2018.

[2] Xianglei Dang, Xiaoyin Wang, Dong Tong, Zichao Xie, Lingda Li, and Keyi Wang. An adaptive filtering mechanism for energy efficient data prefetching. In *In proceedings of the 18th Asia and South Pacific Design Automation Conference, ASP-DAC 2013, Yokohama, Japan, January 22-25, 2013*, pages 332–337, 2013.

[3] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. Coordinated control of multiple prefetchers in multi-core systems. In *In proceedings of the 42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42 2009), December 12-16, 2009, New York, New York, USA*, pages 316–326, 2009.

[4] Brian Hall, Peter Bergner, Alon Shalev Housfater, Madhusudanan Kandasamy, Tulio Magno, Alex Mericas, Steve Munroe, Mauricio Oliveira, Bill Schmidt, Will Schmidt, et al. *Performance optimization and tuning techniques for IBM Power Systems processors including IBM POWER8*. IBM Redbooks, 2017.

[5] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[6] Víctor Jiménez, Alper Buyuktosunoglu, Pradip Bose, Francis P. O'Connell, Francisco J. Cazorla, and Mateo Valero. Increasing multicore system efficiency through intelligent bandwidth shifting. In *In proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015*, pages 39–50, 2015.

[7] M. Li, G. Chen, Q. Wang, Y. Lin, P. Hofstee, P. Stenstrom, and D. Zhou. Pater: A hardware prefetching automatic tuner on ibm power8 processor. *IEEE Computer Architecture Letters*, 15(1):37–40, Jan 2016.

[8] Shih-wei Liao, Tzu-Han Hung, Donald Nguyen, Chinyen Chou, Chiaheng Tu, and Hucheng Zhou. Machine learning-based prefetch optimization for data center applications. In *In proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 56:1–56:10, New York, NY, USA, 2009. ACM.

[9] Gordon E Moore. Cramming more components onto integrated circuits, electronics magazine, 1965.

[10] K.J. Nesbit, A.S. Dhodapkar, and J.E. Smith. Ac/dc: an adaptive data cache prefetcher. In *PACT*, pages 135–145, 2004.

[11] Vicent Selfa Oliver. *Adaptative Prefetching and Cache Partitioning for Multicore Processors*. PhD thesis, 2018.

[12] Cristobal Ortega, Miquel Moretó, Marc Casas, Ramon Bertran, Alper Buyuktosunoglu, Alexandre E. Eichenberger, and Pradip Bose. libprism: an intelligent adaptation of prefetch and SMT levels. In *In proceedings of the International Conference on Supercomputing, ICS 2017, Chicago, IL, USA, June 14-16, 2017*, pages 28:1–28:10, 2017.

[13] Subbarao Palacharla and Richard E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *In proceedings of the 21st Annual International Symposium on Computer Architecture. Chicago, IL, USA, April 1994*, pages 24–33, 1994.

[14] Biswabandan Panda and Shankar Balachandran. Expert prefetch prediction: An expert predicting the usefulness of hardware prefetchers. *Computer Architecture Letters*, 15(1):13–16, 2016.

[15] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent ram. *IEEE micro*, 17(2):34–44, 1997.

[16] Josue Feliu Perez. *Palnificacio multinivell per a optimitzar l'us de l'ample de banda de la jerarquia de memoria en multinuclis*. PhD thesis, 2011.

[17] Vicent Selfa, Julio Sahuquillo, María E. Gómez, and Crispín Gómez. Efficient selective multicore prefetching under limited memory bandwidth. *Journal of Parallel and Distributed Computing*, https://doi.org/10.1016/j.jpdc.2018.05.002.

[18] Balaram Sinharoy, JA Van Norstrand, Richard J Eickemeyer, Hung Q Le, Jens Leenstra, Dung Q Nguyen, B Konigsburg, K Ward, MD Brown, Jose E Moreira, et al. Ibm power8 processor core microarchitecture. *IBM Journal of Research and Development*, 59(1):1–21, 2015.

[19] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights landing: Second-generation intel xenon phi product. *Ieee micro*, 36(2):34–46, 2016.

[20] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *In proceedings of the 13st International Conference on High-Performance Computer Architecture (HPCA-13 2007), 10-14 February 2007, Phoenix, Arizona, USA*, pages 63–74, 2007.

[21] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 63–74. IEEE, 2007.

[22] Marti Torrents Lapuerta. Improving prefetching mechanisms for tiled cmp platforms. 2016.

[23] Julio Antonio Vivas Vivas. *Evaluacion de la jerarquia de cache en procesadores multinucleo*. PhD thesis, 2015.

[24] Xiaotong Zhuang and Hsien-Hsin S. Lee. Reducing cache pollution via dynamic data prefetch filtering. *IEEE Trans. Computers*, 56(1):18–31, 2007.

APÉNDICE A

Código ejecución de las mezclas en C

```c
#include <sys/types.h>
#define _GNU_SOURCE
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdarg.h>
#include <errno.h>
#include <unistd.h>
#include <string.h>
#include <stdarg.h>
#include <sys/wait.h>
#include <sys/ptrace.h>
#include <err.h>
#include <sys/poll.h>
#include <sched.h>

#include "perf_util.h"

#define N_MAX 10
#define PTRACE_DSCR 44

typedef struct {
  char *events;
  int delay;
  int pinned;
  int group;
  int verbose;
} options_t;

typedef struct {
  pid_t pid;
  int benchmark;
  int n_fins;
  int status;
  int core;
  int dscr;
  int current_counter;
  uint64_t counters [45];

  perf_event_desc_t *fds;
  int num_fds;
  cpu_set_t mask;

} node;
```

```
46
47 node queue [N_MAX];
48 static options_t options;
49 int completed_proc;
50 int fin_ejecucion = 0;
51 int N;
52 int carga;
53
54
55 uint64_t pmu_counters[7];
56 char *events[7];
57
58 char *benchmarks[][200] = {
59   // 0 -> perlbench
60   {NULL, NULL, NULL},
61   // 1 -> bzip2
62   {"/home/carnase1/working_dir/spec_bin/bzip2.ppc64", "/home/carnase1/
       working_dir/CPU2006/401.bzip2/data/all/input/input.combined", "200",
       NULL},
63   // 2 -> gcc
64   {"/home/carnase1/working_dir/spec_bin/gcc.ppc64", "/home/carnase1/
       working_dir/CPU2006/403.gcc/data/ref/input/scilab.i", "-o", "scilab.s
       ", NULL},
65   // 3 -> mcf
66   {"/home/carnase1/working_dir/spec_bin/mcf.ppc64", "/home/carnase1/
       working_dir/CPU2006/429.mcf/data/ref/input/inp.in", NULL},
67   // 4 -> gobmk
68   {"/home/carnase1/working_dir/spec_bin/gobmk.ppc64", "--quiet", "--mode",
       "gtp", NULL},
69   // 5 -> hmmer
70   {"/home/carnase1/working_dir/spec_bin/hmmer.ppc64", "--fixed", "0", "--
       mean", "500", "--num", "500000", "--sd", "350", "--seed", "0", "/home
       /carnase1/working_dir/CPU2006/456.hmmer/data/ref/input/retro.hmm",
       NULL},
71   // 6 -> sjeng
72   {"/home/carnase1/working_dir/spec_bin/sjeng.ppc64", "/home/carnase1/
       working_dir/CPU2006/458.sjeng/data/ref/input/ref.txt", NULL},
73   // 7 -> libquantum
74   {"/home/carnase1/working_dir/spec_bin/libquantum.ppc64", "1397", "8",
       NULL},
75   // 8 -> h264ref
76   {"/home/carnase1/working_dir/spec_bin/h264ref.ppc64", "-d", "/home/
       carnase1/working_dir/CPU2006/464.h264ref/data/ref/input/
       foreman_ref_encoder_baseline.cfg", NULL},
77   // 9 -> omnetpp
78   {"/home/carnase1/working_dir/spec_bin/omnetpp.ppc64", "/home/carnase1/
       working_dir/CPU2006/471.omnetpp/data/ref/input/omnetpp.ini", NULL},
79   // 10 -> astar
80   {"/home/carnase1/working_dir/spec_bin/astar.ppc64", "/home/carnase1/
       working_dir/CPU2006/473.astar/data/ref/input/BigLakes2048.cfg", NULL
       },
81   // 11 -> xalancbmk
82   {"/home/carnase1/working_dir/spec_bin/Xalan.ppc64", "-v", "/home/carnase1
       /working_dir/CPU2006/483.xalancbmk/data/ref/input/t5.xml", "/home/
       carnase1/working_dir/CPU2006/483.xalancbmk/data/ref/input/xalanc.xsl"
       , NULL},
83   // 12 -> bwaves
84   {"/home/carnase1/working_dir/spec_bin/bwaves.ppc64", NULL},
85   // 13 -> gamess
86   {"/home/carnase1/working_dir/spec_bin/gamess.ppc64", NULL},
87   // 14 -> milc
88   {"/home/carnase1/working_dir/spec_bin/milc.ppc64", NULL},
89   // 15 -> zeusmp
90   {"/home/carnase1/working_dir/spec_bin/zeusmp.ppc64", NULL},
```

```
91    // 16 -> gromacs
92    {"/home/carnase1/working_dir/spec_bin/gromacs.ppc64", "-silent", "-deffnm
          ", "/home/carnase1/working_dir/CPU2006/435.gromacs/data/ref/input/
          gromacs", "-nice", "0", NULL},
93    // 17 -> cactusADM
94    {"/home/carnase1/working_dir/spec_bin/cactusADM.ppc64", "/home/carnase1/
          working_dir/CPU2006/436.cactusADM/data/ref/input/benchADM.par", NULL
          },
95    // 18 -> leslie3d
96    {"/home/carnase1/working_dir/spec_bin/leslie3d.ppc64", NULL},
97    // 19 -> namd
98    {"/home/carnase1/working_dir/spec_bin/namd.ppc64", "--input", "/home/
          carnase1/working_dir/CPU2006/444.namd/data/all/input/namd.input", "--
          iterations", "38", "--output", "namd.out", NULL},
99    // 20 -> microbench
100   {"/home/carnase1/working_dir/microbenchArray160MBinf", "100", "0", "1024"
          , "0"},
101   // 21 -> soplex
102   {"/home/carnase1/working_dir/spec_bin/soplex.ppc64", "-s1", "-e","-m45000
          ", "/home/carnase1/working_dir/CPU2006/450.soplex/data/ref/input/pds
          -50.mps", NULL},
103   //22 -> povray
104   {"/home/carnase1/working_dir/spec_bin/povray.ppc64", "/home/carnase1/
          working_dir/CPU2006/453.povray/data/ref/input/SPEC-benchmark-ref.ini"
          , NULL},
105   // 23 -> GemsFDTD
106   {"/home/carnase1/working_dir/spec_bin/GemsFDTD.ppc64", NULL},
107   // 24 -> lbm
108   {"/home/carnase1/working_dir/spec_bin/lbm.ppc64", "300", "reference.dat",
           "0", "1", "/home/carnase1/working_dir/CPU2006/470.lbm/data/ref/input
          /100_100_130_ldc.of", NULL},
109   // 25 -> tonto
110   {"/home/carnase1/working_dir/spec_bin/tonto.ppc64", NULL},
111   // 26 -> calculix
112   {"/home/carnase1/working_dir/spec_bin/calculix.ppc64", "-i", "/home/
          carnase1/working_dir/CPU2006/454.calculix/data/ref/input/
          hyperviscoplastic", NULL},
113   // 27
114   {NULL, NULL, NULL},
115 };
116
117 char *benchNames [] = {"perlBench","bzip2","gcc","mcf","gobmk","hmmer","
        sjeng","libquantum","h264ref","omnetpp","astar","xalancbmk","bwaves","
        gamess","milc","zeusmp","gromacs","cactusADM","leslie3d","namd","
        microbench","soplex","povray","gemsFDTD","lbm","tonto","calculix"};
118
119 unsigned long int instruccions_totals [] = {
120   0, 558309327207, 5421059240140, 186483654001,
          0,776504509655,614626187081,
121   480070400876,802635200538,261219407437,428862715907,470328894765,428418848680,

122   886931409787,204234912479,504060702499,463926761740,843934894626,492910117299,

123   498894476043,87430624497,373477511853,628243699177,376876177856,445124040617,0,0,0

124 };
125
126 int nmezclas [] = {8,8,10,10};
127
128 int mezclas [][12] = {
129   {17,15,10,14,3,11,21,9},
130   {17,15,10,11,3,14,23,21},
131   {17,17,15,23,11,10,10,3,9,9},
132   {17,11,21,21,14,14,21,23,23,23}
```

```
};
int def = 0;
int off = 0;
int DSCRBench [] =
    {0,0,0,0,0,0,0,0,0,1,1,66,0,0,66,0,0,0,0,0,0,0,0,0,0,0,0};


static int do_dscr_pid(int dscr_state, pid_t pid)
{
  int rc;

  rc = ptrace(PTRACE_ATTACH, pid, NULL, NULL);
  if (rc) {
    fprintf(stderr, "Could not attach to process %d to %s the "
      "DSCR value\n%s\n", pid, (dscr_state ? "set" : "get"),
      strerror(errno));
    return rc;
  }

  wait(NULL);


    rc = ptrace(PTRACE_POKEUSER, pid, PTRACE_DSCR << 3, dscr_state);
    if (rc) {
      fprintf(stderr, "Could not set the DSCR value for pid "
        "%d\n%s\n", pid, strerror(errno));
      ptrace(PTRACE_DETACH, pid, NULL, NULL);
      return rc;
    }


  rc = ptrace(PTRACE_PEEKUSER, pid, PTRACE_DSCR << 3, NULL);
  if (errno) {
    fprintf(stderr, "Could not get the DSCR value for pid "
      "%d\n%s\n", pid, strerror(errno));
    rc = -1;
  } else {
    printf("DSCR for pid %d is %d\n", pid, rc);
  }

  ptrace(PTRACE_DETACH, pid, NULL, NULL);
  return rc;
}


static void get_countsv2(node *aux){
  ssize_t ret;
  int i;

  for(i=0; i < aux->num_fds; i++) {
    uint64_t val;

    ret = read(aux->fds[i].fd, aux->fds[i].values, sizeof(aux->fds[i].
        values));
    if (ret < (ssize_t)sizeof(aux->fds[i].values)) {
      printf("FALLA ALGO.\n");
      if (ret == -1)
      err(1, "cannot read values event %s", aux->fds[i].name);
      else
      warnx("could not read event%d", i);
    }


    val = aux->fds[i].values[0] - aux->fds[i].prev_values[0];
```

```
195       aux->fds[i].prev_values[0] = aux->fds[i].values[0];
196     aux->counters[i] += val;
197
198     }
199
200 }
201
202 static void get_counts_g(node *aux) {
203     ssize_t ret;
204     int i;
205
206     for(i=0; i < aux->num_fds; i++) {
207       ret = read(aux->fds[i].fd, aux->fds[i].values, sizeof(aux->fds[i].
              values));
208       if (ret < (ssize_t)sizeof(aux->fds[i].values)) {
209         if (ret == -1)
210         err(1, "cannot read values event %s", aux->fds[i].name);
211         else
212         warnx("could not read event%d", i);
213       }
214     }
215
216     for (i=0; i<aux->num_fds; i++) {
217       aux->counters[aux->current_counter+i] = aux->fds[i].values[0];
218     }
219
220     aux->current_counter += aux->num_fds;
221
222
223
224 }
225
226 int measure() {
227
228     int i, ret;
229
230     for (i=0; i<N; i++) {
231       if (queue[i].pid > 0) {
232         kill(queue[i].pid, 18);
233       }
234     }
235     for (i=0; i<N; i++) {
236       waitpid(queue[i].pid, &(queue[i].status), WCONTINUED);
237       if (WIFEXITED(queue[i].status)) {
238       }
239     }
240
241
242     usleep(options.delay*1000);
243
244
245     ret = 0;
246     for (i=0; i<N; i++) {
247       if (queue[i].pid > 0) {
248         kill(queue[i].pid, 19);
249       }
250     }
251
252     for (i=0; i<N; i++) {
253       waitpid(queue[i].pid, &(queue[i].status), WUNTRACED);
254       if (WIFEXITED(queue[i].status)) {
255         ret++;
256         queue[i].pid = -1;
257       }
```

```
258   }
259
260
261   for (i=0; i<N; i++) {
262   get_countsv2(&(queue[i]));
263   }
264
265   return ret;
266
267 }
268
269
270 int lanzar_proceso (node *node) {
271   FILE *fitxer;
272   pid_t pid;
273
274   pid = fork();
275   switch (pid) {
276
277     case -1:
278     exit(-3);
279
280     case 0:
281
282     switch(node->benchmark) {
283
284       case 4:
285       close(0);
286       fitxer = fopen("/home/carnase1/working_dir/CPU2006/445.gobmk/data/ref
              /input/13x13.tst", "r");
287       if (fitxer == NULL) {
288         printf("Error. No se ha podido abrir el fichero arb.tst.\n");
289         return -1;
290       }
291       break;
292
293       case 13:
294       close(0);
295       fitxer = fopen("/home/carnase1/working_dir/CPU2006/416.gamess/data/
             ref/input/h2ocu2+.gradient.config", "r");
296       if (fitxer == NULL) {
297         printf("Error. No se ha podido abrir el fichero h2ocu2+.energy.
                 config.\n");
298         return -1;
299       }
300       break;
301
302       case 14:
303       close(0);
304       fitxer = fopen("/home/carnase1/working_dir/CPU2006/433.milc/data/ref/
              input/su3imp.in", "r");
305       if (fitxer == NULL) {
306         printf("Error. No se ha podido abrir el fichero su3imp.in.\n");
307         return -1;
308       }
309       break;
310
311       case 18:
312       close(0);
313       fitxer = fopen("/home/carnase1/working_dir/CPU2006/437.leslie3d/data/
             ref/input/leslie3d.in", "r");
314       if (fitxer == NULL) {
315         printf("Error. No se ha podido abrir el fichero leslie3d.in.\n");
316         return -1;
```

```
317       }
318       break;
319
320       case 22:
321       close(2);
322       fitxer = fopen("/home/carnase1/working_dir/povray.sal", "w");
323       if (fitxer == NULL) {
324         printf("Error. No se ha podido abrir el fichero povray.sal\n");
325         return -1;
326       }
327       break;
328     }
329
330     execv(benchmarks[node->benchmark][0], benchmarks[node->benchmark]);
331     exit (-2);
332
333     default:
334     usleep(100);
335
336     kill (pid, 19);
337     waitpid(pid, &(node->status), WUNTRACED);
338     if (WIFEXITED(node->status)) {
339       return -2;
340     }
341
342     node->pid = pid;
343
344
345     if (sched_setaffinity(node->pid, sizeof(node->mask), &node->mask) != 0)
          {
346       exit(1);
347     }
348     if(def){
349
350
351     do_dscr_pid(0,node->pid);
352
353   }else if(off){
354      do_dscr_pid(1,node->pid);
355
356     }
357   else{
358
359      do_dscr_pid(DSCRBench[node->benchmark],node->pid);
360
361     }
362     return 1;
363   }
364 }
365
366 void iniciar_eventos(node *node) {
367   int i, ret;
368
369   if(node->counters[1] < instruccions_totals[node->benchmark]){
370     ret = perf_setup_list_events(options.events, &(node->fds), &(node->
          num_fds));
371     if (ret || (node->num_fds == 0)) {
372       exit (1);
373     }
374   }
375
376   else {
377     ret = perf_setup_list_events(options.events, &(node->fds), &(node->
          num_fds));
```

```
378     if (ret || (node->num_fds == 0)) {
379       exit (1);
380     }
381   }
382
383   node->fds[0].fd = -1;
384   for (i=0; i<node->num_fds; i++) {
385     node->fds[i].hw.disabled = 0;
386     node->fds[i].hw.read_format = PERF_FORMAT_SCALE;
387     node->fds[i].hw.pinned = !i && options.pinned;
388     node->fds[i].fd = perf_event_open(&node->fds[i].hw, node->pid, -1, (
          options.group? node->fds[i].fd : -1), 0);
389     if (node->fds[i].fd == -1) {
390       errx(1, "cannot attach event %s", node->fds[i].name);
391     }
392   }
393
394 }
395
396 void finalizar_eventos (node *node) {
397   int i;
398   for(i=0; i < node->num_fds; i++) {
399     close(node->fds[i].fd);
400   }
401   perf_free_fds(node->fds, node->num_fds);
402   node->fds = NULL;
403 }
404
405 void iniciar_contadors (node *node) {
406   int i;
407
408   node->current_counter = 0;
409
410   for (i=0; i<45; i++) {
411     node->counters[i] = 0;
412   }
413 }
414
415 static void usage(void) {
416   printf("usage:\n");
417   printf("d num-> delay entre lecturas.\n");
418   printf("A num-> carga seleccionada.\n");
419   printf("B -> configuracion prefetch siempre off.\n");
420   printf("N -> configuracion prefetch siempre def.\n");
421   printf("E -> stride para microbenchmark.\n");
422   printf("F -> nops para microbenchmark.\n");
423   }
424
425 int main(int argc, char **argv) {
426   int c, i, ret, quantums = 0;
427   int fi_execucio = 0;
428
429   options.delay = 0;
430   N = -1;
431
432   for (i=0; i<N_MAX; i++) {
433     queue[i].benchmark = -1;
434     queue[i].n_fins = 0;
435     queue[i].pid = -1;
436     queue[i].core = -1;
437
438   }
439
```

```
440   while ((c=getopt(argc, argv,"hpgPA:B:a:b:C:c:E:e:F:f:I:i:J:j:K:k:N:d:"))
          != -1) {
441     switch(c) {
442
443       case 'P':
444       options.pinned = 1;
445       break;
446       case 'g':
447       options.group = 1;
448       break;
449       case 'd':
450       options.delay = atoi(optarg);
451       break;
452       case 'h':
453       usage();
454       exit(0);
455
456       case 'A':
457     carga = atoi(optarg);
458     N = nmezclas[carga];
459
460       //Seleccionar cores predefinidos*/
461       queue[0].core = 0;
462       queue[1].core = 8;
463       queue[2].core = 16;
464       queue[3].core = 24;
465       queue[4].core = 32;
466       queue[5].core = 40;
467       queue[6].core = 48;
468       queue[7].core = 56;
469       queue[8].core = 64;
470       queue[9].core = 72;
471       break;
472
473       case 'B':
474       off = 1;
475       break;
476
477       case 'N':
478       def = 1;
479       break;
480
481       case 'E':
482     //Stride
483     benchmarks[20][3] = optarg;
484     break;
485
486     case 'F':
487     //Nop
488     benchmarks[20][2] = optarg;
489     break;
490
491       default:
492       errx(1, "unknown error");
493     }
494   }
495
496   if (N < 0) {
497     return -1;
498   }
499     options.events = strdup("cycles,instructions,LLC-LOADS,LLC-LOAD-MISSES,
          PM_MEM_PREF");
500
501   if (options.delay < 1) {
```

```
502     options.delay = 200;
503   }
504   if(def){
505    for (i=0; i<N; i++) {
506     DSCRBench[queue[i].benchmark] = 0;
507   }
508   }
509   if(off){
510    for (i=0; i<N; i++) {
511     DSCRBench[queue[i].benchmark] = 1;
512   }
513   }
514
515     for (i=0; i<N; i++) {
516     queue[i].benchmark = mezclas[carga][i];
517   }
518
519   for (i=0; i<N; i++) {
520     if (queue[i].benchmark < 0) {
521       return -1;
522     }
523     if (queue[i].core < 0) {
524       return -1;
525     }
526   }
527   for (i=0; i<N; i++) {
528     iniciar_contadors (&(queue[i]));
529   }
530   for (i=0; i<N; i++) {
531     CPU_ZERO(&(queue[i].mask));
532     CPU_SET(queue[i].core, &(queue[i].mask));
533   }
534
535
536
537   if (pfm_initialize() != PFM_SUCCESS) {
538     errx(1, "libpfm initialization failed\n");
539   }
540
541   for(i=0; i<N; i++) {
542
543     lanzar_proceso(&(queue[i]));
544     iniciar_eventos(&(queue[i]));
545
546
547   }
548
549   do {
550
551     ret = measure();
552
553     quantums++;
554
555
556     if (ret) {
557       for (i=0; i<N; i++) {
558
559         if (queue[i].pid == -1) {
560          get_countsv2(&(queue[i]));
561          finalizar_eventos(&(queue[i]));
562          if(queue[i].counters[1] >= instruccions_totals[queue[i].benchmark
                ]){
563             fin_ejecucion = 1;
564             break;
```

```
565
566             }
567             printf("Algun proceso ha acabado antes de completar las
                    instrucciones\n");
568             lanzar_proceso (&(queue[i]));
569             iniciar_eventos (&(queue[i]));
570
571           }
572         }
573       }
574
575       for (i=0; i<N; i++) {
576       if(queue[i].counters[1] >= instruccions_totals[queue[i].benchmark])
577       {
578         fin_ejecucion = 1;
579         break;
580       }
581
582
583     }
584
585
586     } while(!fin_ejecucion);
587
588     for (i=0; i<N; i++) {
589       if (queue[i].pid > 0) {
590         kill(queue[i].pid, 9);
591           finalizar_eventos(&(queue[i]));
592
593       }
594     }
595
596
597     pfm_terminate();
598
599
600
601     for (c=0; c<N; c++) {
602     fprintf(stderr, "Counters: ");
603     fprintf(stderr, "%s ", benchNames[queue[c].benchmark] );
604
605     for(i = 0; i<queue[c].num_fds;i++){
606       fprintf(stderr, "%20"PRIu64" ", queue[c].counters[i] );
607     }
608
609     fprintf(stderr, "\n");
610     }
611
612     return 0;
613 }
```