Infraestructura de simulación para evaluación de estrategias de particionado de caché en procesadores equipados con la tecnología Intel RDT

TRABAJO FIN DE MÁSTER

Jorge Casas Hernán

Dirigido por: Juan Carlos Sáez Alcaide

Colaborador externo de dirección: Adrián García García

Máster en Ingeniería Informática

Facultad de Informática

Universidad Complutense de Madrid

Junio 2019

Calificación: 10

Infraestructura de simulación para evaluación de estrategias de particionado de caché en procesadores equipados con la tecnología Intel RDT

Simulation-based framework for evaluation of cache partitioning strategies on processors equipped with Intel RDT

Memoria de Trabajo Fin de Máster

Jorge Casas Hernán

Dirigido por: Juan Carlos Sáez Alcaide

Colaborador externo de dirección: Adrián García García

Máster en Ingeniería Informática

Facultad de Informática

Universidad Complutense de Madrid

Junio 2019

Calificación: 10

Resumen

Los procesadores multicore constituyen el tipo de arquitectura más extendida en la actualidad, y están presentes en un amplio espectro de plataformas, desde sistemas de alto rendimiento a dispositivos móviles de bajo consumo. A pesar de sus beneficios, la contención que se produce cuando varias aplicaciones compiten por el uso de recursos compartidos del sistema, como el último nivel de caché (*Last-Level Cache* - LLC) o el ancho de banda con memoria principal, puede ocasionar una degradación sustancial de métricas clave del sistema como el rendimiento global o la justicia. Se ha demostrado que la utilización de técnicas de particionado de caché (aplicadas a la LLC) resulta efectiva para mitigar los efectos de la contención por recursos compartidos. Gracias a ciertas tecnologías recientemente adoptadas en procesadores comerciales –como *Intel Resource Director* Technology (RDT)— , hoy en día es posible implementar estrategias de particionado de caché en el sistema operativo, transladando de forma transparente los beneficios del particionado de caché asistido por hardware directamente a las aplicaciones.

En este Trabajo Fin de Máster se ha creado una infraestructura para asistir a los investigadores en el diseño y evaluación de técnicas de particionado de caché sobre procesadores equipados con la tecnología Intel RDT. La infraestructura desarrollada consta de dos componentes. El primer componente es un conjunto de extensiones realizadas en la herramienta de monitorización del rendimiento PMCTrack para explotar la funcionalidad proporcionada por las distintas tecnologías que conforman Intel RDT. Gracias a este soporte, implementado a nivel de kernel en Linux, es posible obtener información del uso de los recursos compartidos por parte de distintas aplicaciones, y aplicar particionado de caché tanto desde espacio de usuario como desde el propio sistema operativo. El segundo componente de nuestra infraestructura es un simulador de particionado de caché que permite comparar fácilmente la efectividad de diferentes algoritmos de particionado de caché con la solución óptima para distintos objetivos de optimización. El simulador –que recibe como entrada información de las aplicaciones recabada *offline* usando el nuevo soporte desarrollado en PMCTrack,– está equipado con un modelo de predicción de la degradación del rendimiento de las aplicaciones que tiene en cuenta el impacto de la contención en la LLC y el causado por la competencia entre aplicaciones por el uso del ancho de banda con memoria. Para determinar la solución óptima, el simulador cuenta con un algoritmo paralelo de ramificación y poda diseñado específicamente para distribuir el cómputo de forma eficiente entre múltiples cores.

Palabras clave: Intel RDT, Intel CAT, Intel MBM, procesadores multicore, particionado de caché, simulación

Abstract

Multicore processors are the most widespread type of architecture today, and are present on a broad spectrum of platforms, from high-performance systems to low-power mobile devices. Despite their benefits, the contention that occurs when multiple applications compete for shared system resources, such as the last-level cache (LLC) or main memory bandwidth, can result in substantial degradation of key system metrics such as overall performance or fairness. The use of cache partitioning techniques (applied to the LLC) has been shown to be effective in mitigating the effects of shared resource contention. Thanks to certain technologies recently adopted in commercial processors –such as *Intel Resource Director* Technology (RDT)–, it is now possible to implement cache partitioning strategies in the operating system, transparently translating the benefits of hardware-assisted cache partitioning directly to applications.

In this Masters' Thesis, a framework has been created to assist researchers in the design and evaluation of cache partitioning techniques on processors equipped with Intel RDT technology. The developed framework consists of two components. The first component is a set of extensions made in the performance monitoring tool PMCTrack, to exploit the functionality provided by the different technologies that make up Intel RDT. Thanks to this support, implemented at kernel level in Linux, it is possible to obtain information on the use of shared resources by different applications, and apply cache partitioning both from user space and from the operating system itself. The second component of our framework is a cache partitioning simulator that allows you to easily compare the effectiveness of different cache partitioning algorithms with the optimal solution for different optimization purposes. The simulator –which receives information from applications collected offline using the new support developed in PMCTrack– is equipped with a prediction model of application performance degradation that takes into account the impact of contention on the LLC caused by competition between applications for the use of memory bandwidth. To determine the optimal solution, the simulator features a parallel branch and bound algorithm specifically designed to efficiently distribute the computation across multiple cores.

Keywords: Intel RDT, Intel CAT, Intel MBM, multicore processors, cache partitioning, simulation

Autorización de difusión y utilización

El abajo firmante, matriculado en el Máster en Ingeniería Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente trabajo de fin de máster: "Infraestructura de simulación para evaluación de estrategias de particionado de caché en procesadores equipados con la tecnología Intel RDT", realizado durante el curso académico 2018-2019 bajo la dirección de Juan Carlos Sáez Alcaide y con la colaboración externa de dirección de Adrián García García en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en internet y garantizar su preservación y acceso a largo plazo.

Jorge Casas Hernán

Agradecimientos

En primer lugar quería agradecer a Juan Carlos Sáez por todo su apoyo y sus enseñanzas. Sin duda alguna, el profesor más importante de toda mi carrera universitaria.

También me gustaría agradecer a Adrián García por la instrumentación del código del simulador desarrollado en este TFM para la extracción de trazas a visualizar con la herramienta Paraver. El análisis de estas trazas ha sido clave para detectar los problemas de desequilibrio en la carga asignada a los distintos procesos, y con ello poder incrementar la escalabilidad del simulador.

Finalmente, también quería agradecer a mi familia y amigos su total e incondicional apoyo, no sólo durante este TFM sino durante toda mi etapa universitaria.

Índice general

Resumen iii

Abstract v

Autorización de difusión y utilización vii

Agradecimientos ix

1 Introducción 1
- 1.1 Objetivos del proyecto y plan de trabajo 2
- 1.2 Estructura de la memoria 4

2 Particionado de la memoria caché 7
- 2.1 Métricas 7
- 2.2 Problema del particionado de caché óptimo 8
- 2.3 Problema del particionado de caché óptimo en clúster 9
- 2.4 Trabajo relacionado 10

3 Implementación del soporte para Intel RDT en PMCTrack 13
- 3.1 Intel Resource Director Technology (RDT) 13
  - 3.1.1 Intel Cache Monitoring Technology (CMT) 13
  - 3.1.2 Intel Memory Bandwidth Monitoring (MBM) 15
  - 3.1.3 Intel Cache Allocation Technology (CAT) 16
- 3.2 Intel RDT en PMCTrack 18
  - 3.2.1 PMCTrack 18
  - 3.2.2 Implementación del soporte para Intel RDT en PMCTrack 21

4 Diseño e implementación del simulador de particionado de caché 29
- 4.1 Diseño del simulador 29
  - 4.1.1 Estructura de datos de entrada 29
  - 4.1.2 Cálculo del Slowdown 30
  - 4.1.3 Modelo de contención del ancho de banda 31
- 4.2 Implementación del simulador 32

4.2.1 Lenguaje y bibliotecas utilizadas en la implementación 32
4.2.2 Búsqueda paralela de la solución óptima 33
4.2.3 Función de poda 37

5 Guía de usuario básica del simulador 39
5.1 Instalación del simulador 39
5.2 Uso básico del simulador 40

6 Análisis experimental 43
6.1 Entorno experimental 43
6.2 Validación del simulador 44
6.3 Efectividad de la poda 47
6.4 Análisis de la escalabilidad 48

7 Conclusiones y trabajo futuro 51
7.1 Conclusiones 51
7.2 Trabajo futuro 52

A Código fuente del proyecto 55
A.1 Soporte para Intel RDT en PMCTrack 55
A.2 Simulador de particionado de caché 55

B Introduction 57
B.1 Project goals and work plan 58
B.2 Masters' Thesis structure 60

C Conclusions and future work 61
C.1 Conclusions 61
C.2 Future work 62

Bibliografía 65

Lista de tablas

6.1 Características de las dos plataformas utilizadas 43

Lista de figuras

1.1 Recursos compartidos entre cores en el procesador Intel Xeon 2620 E5-v4 (*Broadwell-EP*) . . . 2

2.1 Número de posibles soluciones de particionado de una caché de último nivel de 20 vías según el número de aplicaciones. El número de vías y aplicaciones consideradas están basadas en las características de la plataforma A, descrita en la sección 6.1. . . . 9

3.1 Potencial degradación del rendimiento de una aplicación prioritaria por efecto de la compartición de la LLC. . . . 16
3.2 Posible configuración de las particiones de la LLC en un procesador equipado con Intel CAT. . . . 18
3.3 Arquitectura de PMCTrack . . . 20
3.4 Curvas de fallos de LLC por cada 1000 instrucciones retiradas para distintas aplicaciones de SPEC CPU2006 obtenidas en nuestra plataforma experimental. . . . 27
3.5 Degradación en el rendimiento (*slowdown*) observada al variar el tamaño de la LLC (número de vías) para distintas aplicaciones de SPEC CPU2006 ejecutándose en nuestra plataforma experimental. . . . 28

4.1 Árbol del espacio de búsqueda para el problema de particionado de caché óptimo con 3 aplicaciones y 5 vías. . . . 33
4.2 Trazas del framework Paraver obtenidas para una carga de 6 aplicaciones ejecutándose de forma paralela en la plataforma B usando 4 cores. Las tareas de pre-poda están representadas en verde, las ”normales” de B&B en azul, y los periodos inactivos en gris. . . . 35

5.1 Gráfica generada por el simulador, usando el lanzador *./test/sim.py*, con el siguiente comando: `./test/sim.py -s data/metrics.csv data/ws.csv -a equal-part,ucp -f quiet -C`. . . . 41

6.1 Comparativa entre los valores reales y los proporcionados por el simulador para las métricas de STP y Unfairness en la plataforma A, normalizados a los resultados de la estrategia de particionado Equal-Part. . . . 44

6.2 Comparativa entre los valores reales y los proporcionados por el simulador para las métricas de STP y Unfairness en la plataforma A, normalizados a los resultados de la estrategia de particionado Equal-Part. 46
6.3 Tasa de poda (izquierda) y speedup (derecha) obtenidos para los diferentes conjuntos de cargas de trabajo. La etiqueta del eje X, con formato n/objetivo, indica el número de aplicaciones de la carga de trabajo (n) y el objetivo de optimización –rendimiento (STP) o justicia (Unf)–. . . . 47
6.4 Muestra de la escalabilidad para diferentes cargas de trabajo de 6, 7 y 8 aplicaciones. 49
6.5 Trazas de ejecución para la carga de trabajo 51 en la plataforma B, con 28 cores. 50

B.1 Shared resources between cores in the Intel Xeon 2620 E5-v4 processor (*Broadwell-EP*) 58

Capítulo 1

Introducción

Los procesadores multicore (*Chip Multicore Processors* - CMPs) constituyen hoy en día la arquitectura predominante en sistemas de computación de propósito general. A pesar de sus beneficios, estos procesadores plantean una serie de retos para el software de sistema. Uno de los principales desafíos es cómo mitigar los efectos negativos derivados de la contención que se produce cuando las aplicaciones en ejecución compiten por los recursos compartidos entre los cores del sistema [1]. Nótese, que los cores en un sistema CMP no constituyen unidades de procesamiento completamente independientes, sino que suelen compartir el último nivel de caché (*Last-Level Cache* - LLC) y otros recursos de memoria, como el controlador de DRAM [2], [3].

A modo de ejemplo, la figura 1.1 muestra la existencia de un tercer nivel de caché (L3) compartido entre los distintos cores del procesador Intel Xeon 2620 E5-v4, presente en la plataforma experimental usada en este trabajo. Como también se observa en la figura, el chip integra un único controlador de DRAM, compartido entre los cores. Las aplicaciones que se ejecuten simultáneamente en este sistema pueden competir por el espacio en la LLC o por el uso de ancho de banda con memoria, lo cual puede conllevar una degradación sustancial del rendimiento, potencialmente muy desigual entre aplicaciones [2], [4]–[6].

Se ha demostrado que el particionado de la LLC compartida –es decir, la asignación de una partición de la caché de un cierto tamaño a cada aplicación de una carga de trabajo– resulta efectiva para mitigar los efectos de la contención por recursos compartidos [7], [8]. Recientemente, se ha equipado a los procesadores de Intel con soporte hardware para particionado de la caché compartida, gracias a la tecnología Intel Cache Allocation Technology (CAT) [9] –parte de Intel Resource Director Technology (RDT) [10]—. Intel CAT permite que el sistema operativo (SO), hipervisor o monitor de máquina virtual (VMM) distribuya de forma flexible el espacio disponible en la LLC entre distintas aplicaciones o máquinas virtuales que se ejecutan simultáneamente. Un soporte de particionado de caché similar también se ha adoptado en algunos procesadores de alto rendimiento con arquitectura ARM, como Cavium Thunder X [11], [12].

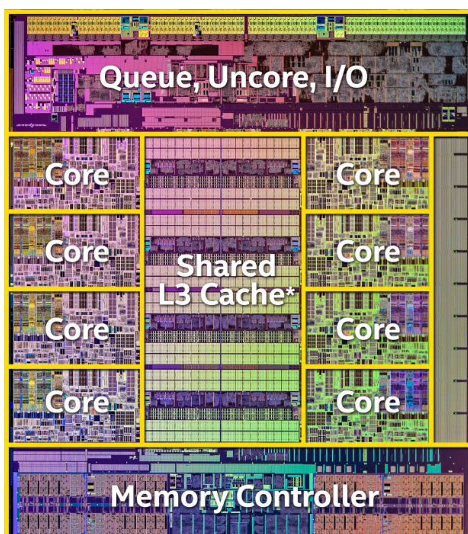


Figura 1.1: Recursos compartidos entre cores en el procesador Intel Xeon 2620 E5-v4 (*Broadwell-EP*)

Hasta la fecha se han propuesto numerosos algoritmos de particionado de caché diseñados para optimizar distintas métricas objetivo [6]–[8], [13], [14]. No obstante, aún no se ha realizado una evaluación exhaustiva de estos algoritmos que indique cuánto se acercan a la solución óptima para distintos objetivos de optimización. Esto hace que resulte complicado cuantificar su verdadero potencial. Como problema adicional, estos algoritmos de particionado de caché no tienen en cuenta de forma precisa la contención derivada de la competencia por ancho de banda, que depende en gran medida del particionado aplicado [13]. Esta fuente adicional de contención puede degradar sustancialmente el rendimiento de cada aplicación individual, reduciendo los beneficios del particionado de caché, haciéndolo incluso contraproducente en ciertas ocasiones [1], [2].

1.1 Objetivos del proyecto y plan de trabajo

El principal objetivo de este Trabajo Fin de Máster ha sido la creación de una infraestructura de investigación que permitiera hacer frente a los problemas anteriormente descritos. Específicamente, la infraestructura ha sido desarrollada para guiar el proceso de diseño de políticas de particionado de caché basadas en la tecnología Intel RDT, y para permitir evaluar rápidamente la efectividad de estas políticas. Esta infraestructura consta de dos componentes: (1) un conjunto de extensiones de la herramienta de monitorización PMCTrack [15] específicas para el uso de Intel RDT en hardware real, y (2) un simulador de particionado de caché.

Las extensiones creadas en PMCTrack proporcionan soporte para particionado de caché y monitorización del ancho de banda consumido por una aplicación. Estas nuevas

características, que ya se han incorporado al repositorio oficial de esta herramienta *open-source* [16], pueden utilizarse tanto desde dentro del sistema operativo (SO) –empleando una API a nivel de kernel– como desde espacio de usuario. En este Trabajo Fin De Master, se ha hecho un uso intensivo de las características accesibles desde espacio de usuario, para recabar los datos de rendimiento experimentales que el simulador desarrollado requiere para funcionar. No obstante, gracias al soporte de SO desarrollado para Intel RDT es posible realizar modificaciones en el kernel Linux para particionar la LLC de forma transparente a los usuarios, y en base a las características de las aplicaciones, para optimizar distintos objetivos. Aunque la implementación de estrategias de particionado de caché en el SO está fuera del alcance de este TFM, ya se han realizado investigaciones donde la infraestructura desarrollada ha sido empleada para este propósito [17].

El simulador de particionado de caché, desarrollado también como parte de este TFM, tiene una doble función. En primer lugar, constituye una herramienta de prototipado de algoritmos de particionado de caché que permite determinar rápidamente (con poco esfuerzo de desarrollo) si un algoritmo concreto es prometedor o no, desde el punto de vista del rendimiento global, la justicia u otras métricas relevantes. En segundo lugar, el simulador permite determinar de manera eficiente el particionado de caché óptimo para diferentes escenarios y objetivos de optimización utilizando un algoritmo paralelo de ramificación y poda, diseñado específicamente para distribuir el cómputo de forma eficiente entre múltiples cores. La capacidad de poder comparar la efectividad de un algoritmo de particionado concreto con la solución óptima es clave para poder evaluar su verdadero potencial.

Para determinar la degradación del rendimiento (*slowdown*) de las aplicaciones de una carga de trabajo multiprogramada –necesario para poder evaluar la efectividad de distintas políticas de particionado–, el simulador está equipado con un modelo de predicción que tiene en cuenta el impacto de la contención en la LLC y el causado por la competencia entre aplicaciones por el uso del ancho de banda con memoria. Para determinar la degradación del rendimiento por contención en la LLC únicamente, el simulador emplea información de rendimiento de aplicaciones (por ejemplo, instrucciones por ciclo) recabada de forma *offline* con los contadores hardware del procesador para distintos tamaños de caché. Para cuantificar el impacto negativo en el rendimiento asociado a la contención en el uso del ancho de banda con memoria, el simulador considera –además de la información obtenida *offline*– el modelo probabilístico propuesto en [18], cuya evaluación requiere la resolución de un sistema de ecuaciones no lineales.

Para proceder al desarrollo y evaluación de la infraestructura, se estableció un plan de trabajo que consta de las siguientes tareas:

- Estudio de la documentación oficial de Intel acerca de la tecnología *Resource Director Technology* (RDT) [10], con especial énfasis en *Cache Allocation Technology* (CAT) y *Memory Bandwidth Monitoring* (MBM).
- Implementación del soporte para Intel RDT en la herramienta PMCTrack.
- Diseño e implementación del simulador en base al modelo de predicción de *slowdown*

descrito anteriormente. En esta fase también se llevó a cabo el diseño de la API para el prototipado y evaluación de algoritmos de particionado.
- Implementación de la versión secuencial del algoritmo de búsqueda del óptimo (ramificación y poda), y diseño de dos funciones de poda asociadas a la optimización del rendimiento global y el grado de justicia del sistema, respectivamente.
- Desarrollo de la versión paralela del simulador para la búsqueda de la solución óptima, distribuyendo de manera eficiente el cómputo entre los cores y tratando de manera efectiva los escenarios de desequilibrio de carga debido a la naturaleza desigual de la poda en diferentes áreas del espacio de búsqueda.
- Evaluación experimental del simulador, incluyendo una validación de los resultados proporcionados por el mismo (comparándolos con los obtenidos en hardware real), y un estudio de rendimiento y escalabilidad del algoritmo paralelo de búsqueda de la solución óptima.

1.2 Estructura de la memoria

El resto de la memoria de este Trabajo Fin de Máster se estructura de la siguiente forma:

- En el **capítulo 2** se introducen las métricas utilizadas para cuantificar el grado de rendimiento global y de justicia de distintas políticas de particionado. También se describe formalmente el problema del particionado de caché óptimo (tanto estricto como en clúster) y se discute el trabajo relacionado.

- En el **capítulo 3** se describen las tecnologías que engloba *Intel Resource Director Technology* (RDT). Posteriormente se presentan las características generales de PMCTrack y se proporcionan detalles sobre la implementación del soporte para Intel RDT desarrollado en esta herramienta.

- En el **capítulo 4** se presentan aspectos de diseño e implementación del simulador de particionado de caché. En este capítulo también se describe en detalle la estructura de los datos de entrada del simulador, la técnica utilizada para aproximar el *slowdown* de cada aplicación al compartir recursos con otras, el algoritmo paralelo de búsqueda de la solución óptima, y las estrategias de poda diseñadas para distintos objetivos de optimización.

- En el **capítulo 5** se proporciona un tutorial básico de uso del simulador, que incluye también instrucciones de instalación.

- En el **capítulo 6** se recoge el análisis experimental realizado con el simulador, que incluye una serie de experimentos de validación del mismo, y el análisis de distintos aspectos del algoritmo de búsqueda de la solución óptima.

- En el **capítulo 7** se presentan las conclusiones finales del trabajo y se proponen

posibles líneas de trabajo futuro.

- Finalmente se proporcionan tres apéndices. En el primero (A) se indica la localización del código del proyecto, y en los otros dos restantes se incluye (B) Introducción y (C) Conclusiones y trabajo futuro, traducidos al inglés.

Capítulo 2

Particionado de la memoria caché

En este capítulo inicialmente se describen las métricas utilizadas para la evaluación de objetivos de optimización, como son la justicia y el rendimiento. Posteriormente se proporcionan detalles acerca del problema del particionado de caché (estricto y en clúster) óptimo. Finalmente se presenta información acerca del trabajo relacionado.

2.1 Métricas

Para medir la degradación del rendimiento que sufre una aplicación ejecutándose en un sistema multicore junto a otras aplicaciones que compiten por el uso de los recursos compartidos del sistema usamos la métrica *Slowdown*. Esta métrica se define de la siguiente manera:

$$Slowdown_{app} = \frac{T_{shared,app}}{T_{alone,app}}$$

donde $T_{shared,app}$ es el tiempo de ejecución de la aplicación ejecutándose junto con el resto de aplicaciones; y $T_{alone,app}$ es el tiempo de ejecución de la aplicación ejecutándose sola en el sistema.

El *slowdown* de una aplicación secuencial (un único hilo de ejecución) puede calcularse también de acuerdo a la media de instrucciones por ciclo de la aplicación ejecutándose sola en el sistema ($IPC_{alone,app}$) con respecto a ejecutarse junto a otras aplicaciones ($IPC_{shared,app}$). La fórmula queda de la siguiente manera:

$$Slowdown_{app} = IPC_{alone,app}/IPC_{shared,app} \tag{2.1}$$

Trabajos previos sobre justicia en sistemas multicore [2], [19] definen un sistema como justo cuando en una carga de trabajo multiprogramada, las aplicaciones de la misma prioridad tienen el mismo *slowdown*. Para aplicar este concepto de la justicia utilizamos la métrica *Unfairness*, la cual ya ha sido utilizada en numerosos trabajos previos [2], [3], [19], [20]. Dada una carga de trabajo de n aplicaciones, esta métrica se define de la siguiente manera:

$$Unfairness = \frac{MAX(Slowdown_1, ..., Slowdown_n)}{MIN(Slowdown_1, ..., Slowdown_n)} \tag{2.2}$$

En cuanto a la medición del rendimiento global de un sistema, en trabajos previos [6], [19], [21] se ha utilizado la métrica *System ThroughPut (STP)*, definida de la siguiente manera para una carga de trabajo de n aplicaciones:

$$STP = \sum_{i=1}^{n} \left(\frac{T_{alone,i}}{T_{shared,i}} \right) = \sum_{i=1}^{n} \left(\frac{1}{Slowdown_i} \right) \tag{2.3}$$

2.2 Problema del particionado de caché óptimo

Sea A una carga de trabajo de n aplicaciones $\{a_1, a_2, \cdots, a_n\}$, y sea S un sistema con una memoria caché de último nivel (LLC) de k vías (*ways*) donde $k \geq n$. El particionado de caché óptimo de la LLC para A en S es definido como el conjunto $\{w_1, w_2, \cdots, w_n\}$ (siendo $\sum_{i=1}^{n} w_i = k$) que alcanza el óptimo de un determinado *target* de optimización (como es la justicia o el rendimiento), donde w_i (siendo $1 \leq i \leq n$) representa el número de vías asignadas a la aplicación a_i ($1 \leq w_i \leq k - n + 1$).

El cálculo del particionado de caché óptimo es considerado como un problema de complejidad NP duro [8]. Para obtener la mejor solución es necesario explorar el espacio de búsqueda de soluciones completo. La siguiente definición recursiva permite obtener el número de posibles soluciones de particionado P para un sistema con una LLC de k vías ejecutando una carga de trabajo de n aplicaciones:

$$P(k, n) = \begin{cases} 1 & k = n, n = 1 \\ n & k = n + 1 \\ \sum_{i=1}^{k-n+1} P(k - i, n - 1) & o.c. \end{cases} \tag{2.4}$$

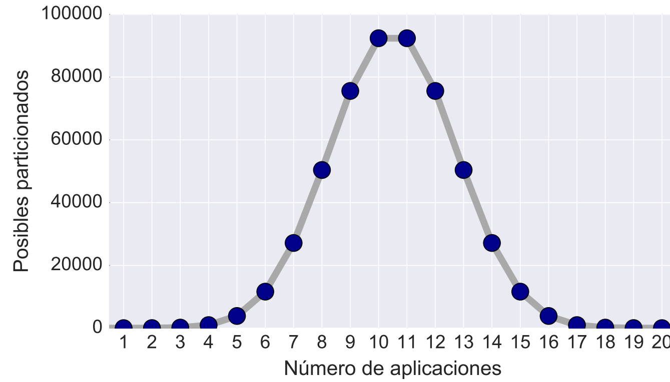


Figura 2.1: Número de posibles soluciones de particionado de una caché de último nivel de 20 vías según el número de aplicaciones. El número de vías y aplicaciones consideradas están basadas en las características de la plataforma A, descrita en la sección 6.1.

Como se puede ver en la figura 2.1, la función P alcanza el valor máximo cuando $n = k/2$. Para $n < k/2$, a medida que aumentamos el número de aplicaciones n, el número de posibles soluciones de particionado aumenta rápidamente, pero va disminuyendo de forma simétrica cuando $n > k/2$.

2.3 Problema del particionado de caché óptimo en clúster

Cuando el número de aplicaciones ejecutándose en el sistema supera el número de vías de la caché ($n > k$) el particionado estricto no puede realizarse, ya que al menos dos aplicaciones tienen que compartir una misma partición. En trabajos previos se ha llegado a la conclusión de que incluso cuando el número de aplicaciones es igual o inferior al número de vías de la caché ($n <= k$) puede resultar ineficaz el particionado de caché estricto (ninguna aplicación comparte una partición con otras) [6], [14]. Esto es debido a la gruesa granularidad del particionado de caché que permiten los procesadores modernos equipados con la tecnología *Intel Cache Allocation Technology* (del orden de MBs). Gracias al particionado de caché óptimo en clúster, podemos incrementar el rendimiento (incluso en ciertas ocasiones también la justicia) debido al aumento de la granularidad, al permitir que dos o más aplicaciones compartan una misma partición [6].

Al igual que en el problema del particionado de caché óptimo, podemos definir formalmente el particionado de caché en clúster óptimo de la siguiente manera. Sea A una carga de trabajo de n aplicaciones $\{a_1, a_2, \cdots, a_n\}$, y sea S un sistema con una LLC de k vías. El particionado en clúster óptimo de la LLC para A en S se define como el conjunto $T = \{C_1, C_2, \cdots, C_m\}$ y su correspondiente conjunto W de vías asignadas

a cada C_i en T, $W = \{w_1, w_2, \cdots, w_m\}$ que alcanza el valor óptimo de un determinada métrica de optimización, donde (1) cada C_i es un subconjunto de A ($C_i \subseteq A$), (2) $1 \leq m \leq \min(n, k)$, (3) $C_1 \cup C_2 \cup \cdots \cup C_m = A$, (4) $\forall i, j, 1 \leq i, j \leq m \wedge j > i, C_i \cap C_j = \emptyset$ y (5) $(1 \leq w_j \leq k - m + 1) \wedge \sum_{i=1}^{m} w_i = k$. Explicado en lenguaje natural, el conjunto T es una posible agrupación de las aplicaciones de la carga de trabajo A en subconjuntos de al menos una aplicación. A cada elemento (subconjunto de aplicaciones) en T lo llamaremos *clúster*. Cada clúster tendrá un número de vías de la caché asignadas, tal y como está especificado en el conjunto W (Por ejemplo, C_1 tiene w_1 vías de la caché asignadas), que representa una posible distribución del espacio en la LLC entre los clústers de aplicaciones de la carga de trabajo.

Teniendo en cuenta el tamaño del espacio de búsqueda, el problema del particionado de caché óptimo en clúster es aún más complejo de resolver que su correspondiente versión estricta. Para determinar la solución óptima recorriendo el espacio de búsqueda completo para cada posible agrupación de aplicaciones (conjunto T) del conjunto A, se debe determinar la distribución de vías correcta entre los clústers de aplicaciones disponibles que optimiza un determinado *target* de optimización para esa agrupación concreta. El número de posibles soluciones crece exponencialmente a medida que aumentamos k y n. En este Trabajo Fin de Máster no se ha considerado una propuesta de solución a este problema, quedando como posible trabajo futuro (tal y como se explica en la sección 7.2).

2.4 Trabajo relacionado

Existen multitud de trabajos previos donde se ha estudiado el problema del particionado de caché y se han propuesto distintas alternativas para acercarse a soluciones óptimas sin necesidad de recorrer el espacio de búsqueda completo, haciendo uso de algoritmos de aproximación [7], [13], [22]. Un estudio reciente [8] analiza las mejores alternativas disponibles para aproximarse a objetivos de optimización, como la maximización del rendimiento, reducir el consumo de energía, o mejorar la justicia. Este TFM se centra en el problema del particionado de caché óptimo, y en la optimización del rendimiento y la justicia. En particular, en el simulador desarrollado se implementan los algoritmos de aproximación UCP [7] y Yu-Petrov [13].

El objetivo de UCP es mejorar el rendimiento global minimizando el número de errores de la LLC, provocados por todas las aplicaciones de la carga de trabajo. UCP no busca la solución óptima, sino que emplea un algoritmo heurístico llamado *lookahead* [7], el cual recibe como entrada la tabla de MPKIs (fallos de caché por cada 1000 instrucciones) de cada aplicación. Esta tabla almacena el valor de MPKI de la aplicación para todos los posibles tamaños de caché (o el número de vías asignadas a la aplicación en el contexto del problema del particionado de caché). En su propuesta inicial [7], UCP se basa en extensiones hardware para determinar la tabla de MPKIs en tiempo de

ejecución. Sin embargo, habiendo pasado más de una década desde su propuesta, estas extensiones hardware no han sido aún incluidas en plataformas comerciales. Para medir el rendimiento y la justicia de UCP, el simulador se basa en dicha tabla de MPKIs con los datos ya calculados.

En [13] Yu y Petrov proponen un algoritmo cuyo objetivo es reducir el uso del ancho de banda del sistema. Para ello, el algoritmo emplea una tabla similar a la de UCP, con la diferencia de que, en vez de almacenar los MPKIs de cada aplicación de la carga de trabajo para los distintos tamaños de la caché, se almacenan las distintas medidas del uso del ancho de banda.

Una de las principales aportaciones de este TFM es el uso de un algoritmo paralelo de ramificación y poda (*Branch and Bound* - B&B) con el que cuenta el simulador para determinar la solución óptima del particionado de caché. La paralelización de los algoritmos B&B ha sido objeto de estudio desde hace ya mucho tiempo, siendo uno de los problemas clásicos de la computación [23]–[25], habiendo sido propuestas una gran cantidad de variantes del algoritmo de B&B [23]. Dejando de lado los detalles técnicos específicos de la implementación, existen una serie de aspectos que se tienen que tener en cuenta ya que pueden provocar cuellos de botella derivados de la paralelización. En primer lugar, las tareas se crean en tiempo de ejecución y su tamaño a priori es desconocido, lo que puede llevar a un desequilibrio de la carga de trabajo entre los cores. En segundo lugar, la asignación de tareas a cada core se debe hacer de forma dinámica pudiendo variar dependiendo de los datos de entrada. Por último, la sobrecarga derivada de las comunicaciones puede ser también un problema para el balanceo de la carga.

Cuando se diseña un algoritmo B&B hay una serie de retos con los que hay que lidiar, que se analizan en detalle en [24]. Algunos de los más importantes son: la definición del espacio de búsqueda, la estrategia de reparto de tareas entre cores, la comunicación entre cores, la minimización del tiempo en el que hay cores ociosos, y la maximización de la computación útil (evitar explorar soluciones que podrían haber sido descartadas mediante una buena política de poda).

En [25] se proporciona un análisis de las posibles partes del algoritmo de B&B que pueden ser paralelizadas. Por un lado, es posible paralelizar la exploración del árbol de búsqueda, dividiendo el espacio de búsqueda en subproblemas donde se definen espacios de búsqueda disjuntos que pueden explorarse de forma independiente al mismo tiempo. Por otro lado, la implementación de una determinada política de poda puede tener un coste computacional alto, por lo que puede interesar paralelizarlo. El algoritmo B&B usado en el simulador de este TFM reune las ventajas de las dos posibles estrategias de paralelización comentadas, mediante el uso de distintas fases del algoritmo para maximizar los beneficios.

A pesar de que la paralelización de algoritmos de B&B ha sido objeto de estudio en numerosos trabajos previos, no se han encontrado trabajos previos que hayan propuesto un algoritmo paralelo de B&B como solución al problema del particionado de caché

óptimo. Los aspectos clave del algoritmo paralelo de B&B del simulador propuesto son: (1) el algoritmo heurístico usado para determinar de forma eficiente la solución inicial que se usará para la poda, (2) el mecanismo para obtener una solución ideal (límite superior e inferior para las métricas STP y $Unfairness$ respectivamente) siendo posible calcularla desde cualquier punto del espacio de búsqueda, y (3) la fase de prepoda, útil para minimizar los efectos impredecibles derivados de la poda.

Capítulo 3

Implementación del soporte para Intel RDT en PMCTrack

En este capítulo se introducen las distintas tecnologías que forman parte de Intel RDT y se describe el soporte introducido en PMCTrack como parte de este TFM para emplear las extensiones de Intel RDT desde espacio de usuario y desde el sistema operativo.

3.1 Intel Resource Director Technology (RDT)

Intel Resource Director Technology (Intel RDT) es un conjunto de tecnologías de los procesadores actuales de Intel que proporcionan capacidades de monitorización -como las tecnologías Cache Monitoring Technology (Intel CMT) y Memory Bandwidth Monitoring (Intel MBM)- y capacidades de asignación de recursos compartidos, como la tecnología Cache Allocation Technology (Intel CAT). Estas tecnologías han existido de forma independiente hasta que recientemente Intel decidió unificarlas bajo las siglas RDT.

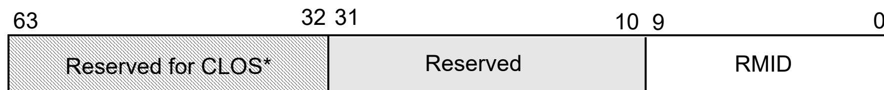
En las siguientes subsecciones se presentan las distintas tecnologías que conforman Intel RDT.

3.1.1 Intel Cache Monitoring Technology (CMT)

La tecnología Intel CMT permite a un sistema operativo o a un Hipervisor/Virtual Machine Monitor (VMM) determinar el uso actual de un determinado nivel de la jerarquía de memoria caché (normalmente es en el último nivel, LLC) de un conjunto de aplicaciones ejecutándose simultáneamente en el sistema. Intel CMT se introdujo por primera vez en la familia de procesadores Intel Xeon E5 v3.

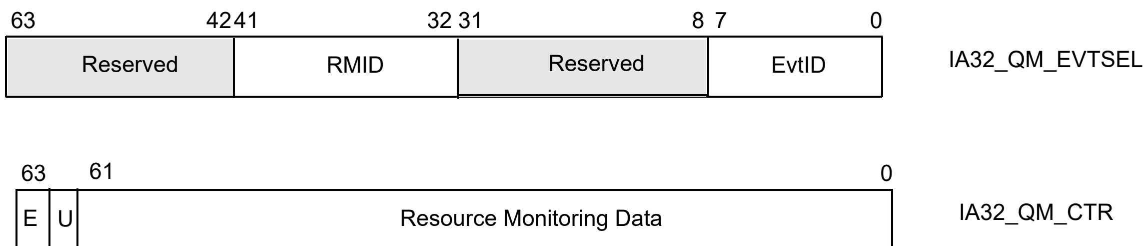
A alto nivel, Intel CMT funciona de la siguiente manera. El sistema operativo o VMM asigna un determinado identificador a cada aplicación/máquina virtual; este identificador se conoce como *Resource Monitoring ID* (RMID). El hardware compatible con la tecnología CMT monitoriza el espacio usado en la LLC por cada RMID, de forma que el sistema operativo o el VMM son capaces de leer la ocupación de la LLC para una determinada aplicación o máquina virtual en cualquier instante.

Para que sea posible hacer un seguimiento de la ocupación de la LLC por aplicación, el procesador debe conocer el RMID de cada hilo (o CPU virtual) que se está ejecutando actualmente en cada core del sistema. Para lograr esto, el hardware expone un registro privilegiado por cada core para almacenar el RMID asociado con el hilo que se está ejecutando actualmente en el core en cuestión. Dicho registro es el `IA32_PQR_ASSOC` y sus campos se muestran en la siguiente figura, extraída de [10]:



Intuitivamente, al producirse un cambio de contexto, el sistema operativo debe actualizar el campo `RMID` (bits 0-9) de este registro con el valor de RMID asignado al hilo de ejecución que entra a ejecutar en esa CPU o core. Los 32 bits más significativos del registro están reservados para la gestión de las tecnologías MBM y CAT, que se describen en las siguientes secciones.

En cualquier instante el sistema operativo o el VMM puede consultar el espacio usado en la LLC para cualquier aplicación en ejecución, empleando los registros `IA32_QM_EVTSEL` e `IA32_QM_CTR`, cuyos campos se muestran a continuación:



Para consultar el espacio en la LLC usado por una aplicación es necesario realizar dos pasos de forma secuencial. En primer lugar debe escribirse su RMID y el valor 0 (EvtID para *cache occupancy*) en los campos correspondientes del registro `IA32_QM_EVTSEL`. A continuación, el espacio usado en caché (indicado en bytes) puede obtenerse leyendo del registro `IA32_QM_CTR`, y quedándose con los 62 bits menos significativos del registro (campo *Resource Monitoring Data*).

Es importante destacar que todos los registros privilegiados que se presentan en este capítulo pertenecen a la categoría de MSRs (*Model Specific Registers*) [26] a los que

el sistema operativo puede acceder mediante las instrucciones máquina `wrmsr` y `rdmsr` [27]. La dirección de acceso a cada uno de los registros, usada como argumento de las instrucciones máquina, puede encontrarse en [10]. En nuestra implementación del soporte para las tecnologías CAT y MBM en PMCTrack (ya existía soporte para CMT antes de la realización de este TFM) accedemos a los MSRs necesarios mediante funciones envoltorio proporcionadas por el kernel Linux, que hacen uso de ensamblador en línea (*inline assembly*) para poder ejecutar las citadas instrucciones privilegiadas desde código C.

3.1.2 Intel Memory Bandwidth Monitoring (MBM)

La tecnología Intel MBM fue construida sobre la infraestructura de Intel CMT e introducida por primera vez en la familia de procesadores Intel Xeon E5 v4. Permite la monitorización del ancho de banda que se consume al transferir datos entre un nivel de la jerarquía de memoria y el siguiente. Los procesadores actuales de Intel, no obstante, solo soportan la monitorización del ancho de banda entre la LLC y la memoria principal.

Al haber sido construido sobre los fundamentos de la tecnología Intel CMT, la labor del software del sistema es la misma: asignación de RMIDs a los procesos y actualización de éstos en los cambios de contexto. El funcionamiento de los Resource Monitoring IDs (RMIDs) es el mismo, por lo que gracias a esta tecnología somos capaces de monitorizar el ancho de banda por cada aplicación/máquina virtual.

En cualquier instante el sistema operativo o el VMM puede consultar el ancho de banda con memoria para cualquier aplicación en ejecución, empleando los registros `IA32_QM_EVTSEL` e `IA32_QM_CTR`, presentados en la sección anterior. En particular, para realizar esta consulta, el software de sistema ha de escribir el RMID de la aplicación/hilo y los valores 1 o 2 en los campos `RMID` y `EvtID` de este registro. A continuación, el ancho de banda (indicado en bytes) puede obtenerse leyendo del registro `IA32_QM_CTR`, y quedándose con los 24 bits menos significativos del registro. Si usamos EvtID=1 al seleccionar el evento en el registro `IA32_QM_EVTSEL`, el valor de ancho de banda reportado será el que corresponde al tráfico con el controlador de memoria local, y para EvtID=2 se obtendrá el tráfico con cualquier controlador de memoria (tanto local como remoto). En nuestra plataforma experimental, solo hay un nodo de memoria, y por tanto un único controlador de memoria, por lo que el valor de los dos eventos de monitorización coincide.

Cabe destacar que el espacio usado en la LLC, que se obtiene al leer de `IA32_QM_CTR` mediante el procedimiento descrito en la sección anterior, proporciona un valor instantáneo del espacio usado (en base a las líneas de caché ocupadas en ese momento por la aplicación). Por el contrario, el ancho de banda reportado por Intel MBM es realmente una cuenta acumulativa del ancho de banda consumido por la aplicación a lo largo del tiempo. Esta cuenta, que se implementa internamente con un contador de

24 bits puede llegar a desbordarse. Por este motivo, en nuestra implementación del soporte correspondiente en PMCTrack, se mantienen dos contadores por cada hilo de ejecución (uno para el ancho de banda local y otro para el global) almacenados en el descriptor de hilo en el kernel, que registran el último valor leido para cada evento. De este modo, PMCTrack puede detectar situaciones de desbordamiento, y medir el ancho de banda consumido por la aplicación en un determinado intervalo de tiempo (frecuencia de muestreo configurable por el usuario).

3.1.3 Intel Cache Allocation Technology (CAT)

El objetivo fundamental de Intel CAT [10] es permitir al software de sistema –sistema operativo o Virtual Machine Monitor (VMM)– asignar un determinado espacio de la caché (especificado en número de vías) a una aplicación en función de la prioridad de la misma mediante un sistema de clases de servicio. Como se ilustra en la figura 3.1, una aplicación de alta prioridad podría ver reducido sustancialmente el espacio que tiene disponible en la LLC, al competir por espacio en caché con otra aplicación que tenga una mayor tasa de demanda (p.ej., un número mayor de fallos de caché por ciclo). El hecho de que el hardware no sea consciente de las prioridades de las aplicaciones [3], [4], [28] -establecidas por el usuario- puede originar esta situación de injusticia en la distribución del espacio en la LLC, lo cual puede afectar negativamente al rendimiento de las aplicaciones de alta prioridad. El particionado de caché por vías, que implementa Intel CAT, permite que el sistema operativo pueda mitigar los efectos negativos de la contención en la LLC [8].

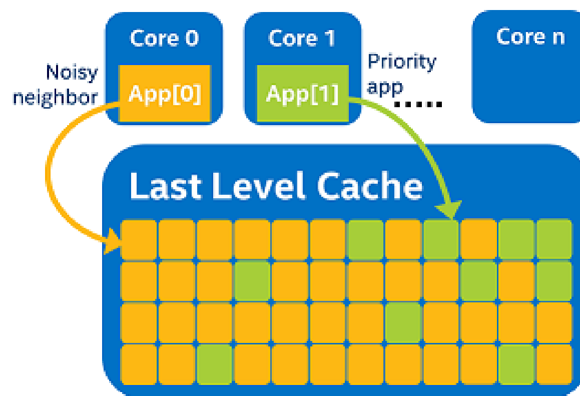


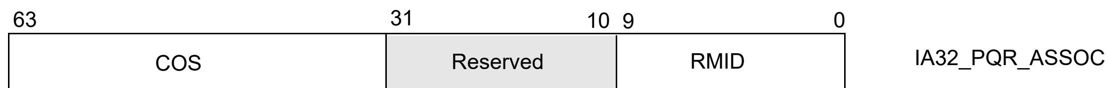
Figura 3.1: Potencial degradación del rendimiento de una aplicación prioritaria por efecto de la compartición de la LLC.

Intel CAT se introdujo por primera vez en la familia de procesadores Intel Xeon E5 v4 (y en un subconjunto de la familia Intel Xeon E5 v3). Existen dos variantes de esta tecnología: CAT1, centrada en la caché exclusivamente de datos; y CAT2, para particionado de caché para instrucciones/datos. En este Trabajo Fin de Máster se ha

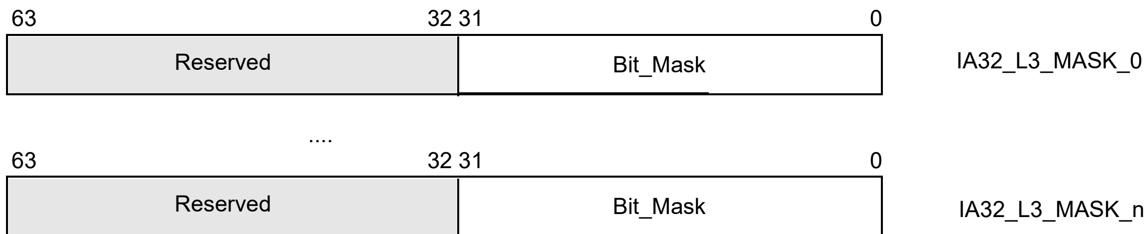
hecho uso de la variante CAT1, y se ha implementado el soporte correspondiente para esta variante en PMCTrack. Los procesadores actuales de Intel sólo implementan CAT para gestionar la distribución de espacio en el último nivel de memoria caché (LLC), pero Intel ya ha anunciado que se incluirá soporte para gestionar el nivel L2 en futuras familias de procesadores [10].

El procesador expone al software de sistema un conjunto finito de clases de servicio (en inglés *Class of Service*, COS) que se pueden asignar a aplicaciones, máquinas virtuales o a hilos individuales. La asignación de espacio en la LLC para las respectivas aplicaciones se restringe en función de la clase a la que están asociadas. Cada COS se puede configurar utilizando máscaras de bits de capacidad (en inglés *Capacity Bitmask*, CBM), que representan el conjunto de vías de la caché de último nivel que conforman una partición de caché.

Para que el software de sistema pueda indicar al hardware el COS asociado a una determinada aplicación en ejecución, el sistema operativo ha de actualizar en cada cambio de contexto el campo `COS` del registro `IA32_PQR_ASSOC` (bits 32-63):



Para especificar la CBM (máscara de vías permitidas en la LLC) asociadas a cada COS el procesador expone al software de sistema un conjunto de registros `IA32_L3_MASK_x`, donde `x` varía entre 0 y n, siendo $n + 1$ el número de COS disponibles en la plataforma:



Como se puede observar en la figura, la máscara de bits (CBM) puede ser de hasta 32 bits –los 32 bits más significativos de cada uno de los registros `IA32_L3_MASK_x` están reservados. La anchura concreta de las CBMs en una arquitectura determinada, así como el número de COS disponibles puede obtenerse usando la instrucción `cpuid`, tal y como se indica en [10]. Aunque en nuestra plataforma experimental empleamos un sistema de un solo socket (una única LLC), es importante destacar que existen tantos conjuntos de registros `IA32_L3_MASK_[0-n]` como LLCs haya en la plataforma. La CPU o core desde donde se ejecute la instrucción de acceso a estos MSRs determina la instancia del registro a la que se accede. Desde el código del sistema operativo se pueden modificar registros de Intel CAT para una LLC remota utilizando el mecanismo de IPIs (*Inter-Processor Interrupts*).

En la figura 3.2 se muestra una posible configuración de las particiones de caché para un procesador dotado de 4 COS (o 4 CLOS, como se denota en el ejemplo) que soporta CBMs de 20 bits (LLC de 20 vías). En el ejemplo, se crean 4 particiones distintas, de diferente tamaño: 4 vías para CLOS 0 y 1, 5 vías para CLOS 2, y 10 para CLOS 3. Como se puede observar, solo hay solapamiento entre la partición para CLOS 2 y CLOS 3. Este solapamiento está permitido; la única restricción que aplica a las CBMs es que deben estar formadas por grupos consecutivos de 1s, es decir, las vías que conforman una partición de caché han de ser siempre contiguas.

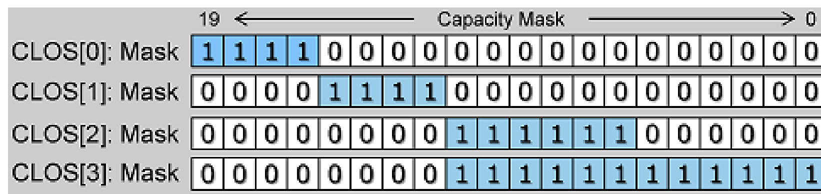


Figura 3.2: Posible configuración de las particiones de la LLC en un procesador equipado con Intel CAT.

3.2 Intel RDT en PMCTrack

En esta sección se realiza una introducción a la herramienta PMCTrack, haciendo especial énfasis en sus características generales y en su arquitectura. A continuación se describe la implementación del soporte de Intel RDT en PMCTrack desarrollado en este TFM.

3.2.1 PMCTrack

PMCTrack es una herramienta *open-source* para Linux que permite la monitorización del rendimiento mediante el uso de contadores hardware [15]. Inicialmente fue desarrollada para facilitar a los investigadores la implementación de algoritmos de planificación en el kernel Linux que hagan uso de información extraída de contadores de monitorización del rendimiento (*Performance Monitoring Counters* - PMCs) para realizar optimizaciones en tiempo de ejecución. Posteriormente se añadieron a PMCTrack distintos componentes de espacio de usuario que permiten al usuario final obtener información de monitorización de aplicaciones en tiempo de ejecución.

Trabajos previos [15], [29] explican en detalle las ventajas de PMCTrack con respecto a otras herramientas de monitorización del rendimiento. PMCTrack soporta la monitorización tanto de aplicaciones secuenciales como paralelas y tiene capacidad de multiplexación de eventos.

La herramienta `pmctrack` de línea de comandos es la forma más directa que tiene el usuario de hacer uso de las funcionalidades de PMCTrack. Este comando cuenta con tres modos de uso:

- *Time-Based Sampling* (TBS): Este modo permite al usuario obtener los valores de los PMCs y contadores virtuales cada cierto intervalo de tiempo (intervalo de muestreo), para una aplicación específica.
- *Event-Based Sampling* (EBS): Este modo permite al usuario obtener los valores de los PMCs y contadores virtuales para una aplicación específica cada vez que un contador determinado alcanza cierto umbral indicado por el usuario.
- *Time-Based system-wide monitoring mode*: Este modo es una variante del modo TBS. La diferencia entre los dos modos radica en que en este modo la información de monitorización se proporciona por cada CPU (core) del sistema, en vez de para una aplicación específica. Este modo se activa mediante la opción -S de `pmctrack`.

Para ilustrar el funcionamiento del comando `pmctrack` consideremos el siguiente ejemplo:

```
$ pmctrack -c instr,cycles ./galgel00
[Event-to-counter mappings]
pmc1 = instr
pmc2 = cycles
[Event counts]
nsample pid     event  pmc1         pmc2
1       27204   tick   2247040326   1723742839
2       27204   tick   2423705061   1957082929
3       27204   tick   2466664612   1944385684
4       27204   tick   2280669757   1964700454
5       27204   tick   2595158266   1983839065
6       27204   tick   2462262543   1980869030
7       27204   tick   2474564037   1942807991
8       27204   tick   2307308640   1978270869
...
```

Este comando proporciona al usuario el número de instrucciones retiradas y los ciclos del procesador por segundo (se usa por tanto TBS, que es el modo por defecto). Tal como se ve en el ejemplo, la opción -c permite la especificación de eventos hardware a monitorizar, pudiendo usar mnemotécnicos como en otras herramientas orientadas a espacio de usuario como `perf`. Al final de la línea se especifica el comando para la ejecución de la aplicación a monitorizar (por ejemplo ./galgel00).

La salida del comando está dividida en dos secciones: La sección *Event-to-counter mappings* muestra la asignación de cada evento hardware a cada contador físico; y la sección *Event counts* muestra en una tabla la información de cada muestra extraída (por defecto una por segundo, aunque el intervalo de muestreo puede configurarse con la

opción -T) representada cada una en una fila diferente de la tabla.

3.2.1.1 Arquitectura y componentes de PMCTrack

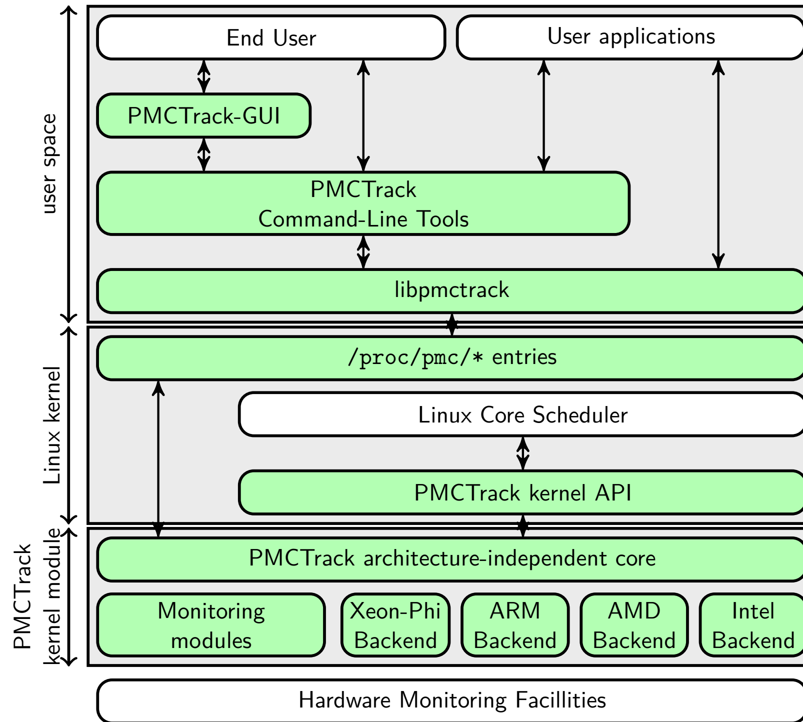


Figura 3.3: Arquitectura de PMCTrack

En la figura 3.3 se muestra la arquitectura de PMCTrack, donde pueden distinguirse componentes en espacio de usuario y en el kernel. El módulo del kernel, que se observa en la parte inferior de la figura, implementa la mayor parte de la funcionalidad de PMCTrack, y se encarga de recopilar — en base a las peticiones expresas del planificador del SO o del usuario— información proporcionada por el hardware de monitorización disponible en la plataforma. Para poder recabar esta información de forma independiente para cada proceso o hilo que se ejecuta en el sistema, el módulo del kernel ha de ser consciente de una serie de eventos que suceden durante el ciclo de vida de un hilo, como los cambios de contexto, los bloqueos por E/S o sincronización, etc. En la actualidad el kernel Linux no ofrece una API para capturar este tipo de eventos desde los módulos cargables, por lo que es preciso añadir este tipo de soporte en el kernel Linux mediante un parche. Este parche, que es muy portable y fácil de adaptar a nuevas versiones del kernel, se representa en la figura 3.3 con el nombre de “PMCTrack kernel API”. La nueva API no sólo permite al módulo del kernel de PMCTrack recibir las notificaciones necesarias (*callbacks*) para la gestión de los recursos hardware de monitorización, sino que también implementa un conjunto de funciones para que cualquier clase de planificación del kernel

Linux pueda acceder a datos de monitorización empleando un mecanismo independiente de arquitectura [15].

El usuario final puede utilizar los servicios de PMCTrack empleando las herramientas de línea de comandos disponibles, la biblioteca *libpmctrack* (que permite instrumentar programas C/C++ para monitorizar el rendimiento de fragmentos de código específicos) o mediante PMCTrack-GUI. Todos estos componentes de espacio de usuario obtienen información de monitorización y configuran los recursos hardware destinados para este propósito mediante un conjunto de ficheros especiales exportados por el módulo del kernel a través del pseudosistema de ficheros `/proc` exportadas por el módulo.

La funcionalidad de PMCTrack puede ampliarse fácilmente mediante la creación de los denominados *módulos de monitorización –plugins* cuya implementación reside en el módulo del kernel de PMCTrack (ver figura 3.3). Gracias a estos módulos es posible exponer al planificador del sistema operativo y al usuario cualquier tipo de información de monitorización proporcionada por los procesadores modernos, pero que no está disponible mediante la interfaz de acceso a los contadores hardware (*Performance Monitoring Unit* o PMU). El consumo de energía o el espacio que una aplicación utiliza en una caché compartida son ejemplos de información que se puede exponer al usuario mediante módulos de monitorización. Para hacer visible esta información a los distintos componentes de PMCTrack se emplean los denominados *contadores virtuales*, que, a diferencia de los contadores hardware "convencionales" no ofrecen datos de la PMU.

3.2.2 Implementación del soporte para Intel RDT en PMCTrack

El soporte para utilizar las extensiones de las tecnologías Intel CAT e Intel MBM, que junto con Intel CMT completan la famila de tecnologías de Intel RDT, se ha implementado en PMCTrack mediante un módulo de monitorización. En lugar de crear un nuevo módulo de monitorización con dicho soporte se ha procedido a modificar el módulo existente que ya tenía soporte para Intel CMT. Las modificaciones de PMCTrack realizadas en este TFM ya se han incorporado al repositorio oficial de la herramienta [16], disponible públicamente.

El código desarrollado, que forma parte del módulo del kernel de PMCTrack, ofrece dos interfaces: (1) una API a nivel de kernel para usar Intel RDT y (2) extensiones de monitorización y asignación de recursos compartidos desde espacio de usuario. Las dos interfaces ofrecen las siguientes funcionalidades:

- Monitorización del ancho de banda (entre la LLC y memoria principal) consumido por una aplicación, tanto local (controlador de memoria local) como global (tráfico entre la LLC y cualquier nodo de memoria de la plataforma)
- Creación de particiones del último nivel de caché asociadas a cada clase de servicio

(COS) disponible. Cada partición se crea asignando una máscara de bits o CBM (ver sección 3.1.3), donde cada 1 de la máscara indica que la vía correspondiente de la LLC puede utilizarse en la partición.

- Asignación de COS a cada proceso[1]

3.2.2.1 API de kernel

El API de kernel de PMCTrack para Intel RDT permite utilizar el soporte hardware de MBM, CAT y CMT desde cualquier módulo de monitorización. Los tipos de datos básicos de la API son los siguientes:

```
#define CMT_MAX_EVENTS      3   /* L3_USAGE, L3_TOTAL_BW, L3_LOCAL_BW */

typedef struct {
        unsigned int cat_nr_cos_available;
    unsigned int cat_cbm_length;
    unsigned int cat_cbm_mask;
} intel_cat_support_t;

typedef struct {
    unsigned int rmid;
    uint64_t last_llc_utilization[RMID_MAX_LLCS][CMT_MAX_EVENTS];
    uint64_t last_cmt_value[RMID_MAX_LLCS][CMT_MAX_EVENTS];
    unsigned int cos_id;
} intel_cmt_thread_struct_t;
```

Esencialmente, para llevar a cabo la activación global de Intel CAT, es preciso obtener un descriptor representado con el tipo de datos `intel_cat_support_t`, que no es más que una estructura que almacena las características de la implementación de CAT específicas del procesador, como el número de clases de servicio (COS) disponibles –`cat_nr_cos_available` o la anchura de las CBMs –`cat_cbm_length`–. Adicionalmente, como parte de las modificaciones realizadas en la infraestructura de monitorización existente se alteró la estructura `intel_cmt_thread_struct_t`, que aloja los datos de monitorización por hilo (visible por el kernel) para Intel RDT. Básicamente, la modificación realizada consistió en convertir los campos `last_cmt_value` y `last_llc_utilization` en arrays de 3 elementos, para almacenar los tres eventos de monitorización actualmente exportados por Intel CMT y MBM: (1) espacio usado en la LLC, (2) ancho de banda local y (3) ancho de banda global. El campo `last_llc_utilization` almacena el último valor del evento en cuestión leído en el punto de monitorización anterior, y `last_cmt_value` el valor leído en el punto de monitorización actual. La implementación realizada emplea ambos contadores para proporcionar el valor del evento registrado entre dos puntos de

[1]En el caso de no asignar COS de forma explícita a un proceso, PMCTrack asignará el COS 0 al mismo.

monitorización consecutivos, que dependen del intervalo de muestreo configurado por el usuario o por el administrador del sistema.

Las funciones que conforman el API del kernel son las siguientes:

```
int intel_cat_initialize(intel_cat_support_t* cat_support);
int intel_cat_release(intel_cat_support_t* cat_support);
int intel_cat_print_capacity_bitmasks(char* str, intel_cat_support_t* cat_support);
int intel_cat_set_capacity_bitmask(intel_cat_support_t* cat_support, unsigned int idx,
                           unsigned int mask);
void __set_rmid_and_cos(unsigned int rmid, unsigned int cosid);
u64 __rmid_read(unsigned long rmid, unsigned int event);
void intel_cmt_update_supported_events(intel_cmt_support_t* cmt_support,
                           intel_cmt_thread_struct_t* data, unsigned int llc_id);
```

Antes de comenzar a usar la tecnología Intel CAT, debe invocarse la función `intel_cat_initialize()`, que devuelve un descriptor (parámetro por referencia). Si la API se utilizara desde el propio kernel Linux, por ejemplo desde el planificador de procesos, esta función se ha de invocar en la rutina de inicialización del planificación, que se ejecuta durante el proceso de arranque del SO. En los módulos de monitorización, `intel_cat_initialize()` debe invocarse desde su función de inicialización[2]. Análogamente, al terminar de usar las extensiones de CAT debe invocarse la función `intel_cat_release()`. El descriptor devuelto por `intel_cat_initialize()` se ha de pasar como parámetro a las funciones que permiten leer el valor de las máscaras establecidas (`intel_cat_print_capacity_bitmasks()`) o modicar la máscara CBM de una COS concreta (`intel_cat_set_capacity_bitmask()`).

El resto de funciones de la API sirven para obtener información de monitorización del espacio usado en caché (Intel CMT) o de ancho de banda consumido (intel MBM). En particular para establecer el COS y RMID asignado[3] a un proceso, al producirse un cambio de contexto (proceso entrante) se debe invocar la función `__set_rmid_and_cos()`. La función `__rmid_read()` permite leer el espacio en caché (`event`=1), o el ancho de banda local o global (valores 2 o 3 para parámetro `event`) asociado a un RMID concreto. Por último, la función `intel_cmt_update_supported_events()` permite actualizar la estructura por hilo (parámetro `data`) con los valores de los eventos asociados al RMID asignado al hilo actual; esta función usa `__rmid_read()` internamente.

En este Trabajo Fin De Master, el uso de esta API se ha restringido al módulo de monitorización que permite llevar a cabo la monitorización y asignación de recursos compartidos desde espacio de usuario. No obstante, gracias a este nuevo soporte es posible crear extensiones del kernel Linux que permitan particionar la LLC de forma transparente a los usuarios, y en base a las características de las aplicaciones. Diseñar estrategias de

[2]La API de los módulos de monitorización está ampliamente documentada en [30]

[3]En esta memoria se omite la descripción de las funciones del API de PMCTrack para la asignación automática RMIDs, ya que estas funciones ya estaban disponibles en PMCTrack antes de la realización de este TFM

particionado de caché a nivel de sistema operativo que hagan uso de esta API constituyen una línea interesante de trabajo futuro. En la actualidad, la implementación de estas políticas de particionado se podría prototipar mediante módulos de monitorización de PMCTrack, lo cual permitiría realizar desarrollos a nivel de kernel que no obliguen a reiniciar el sistema tras cada modificación realizada. Al fin y al cabo, los módulos de monitorización forman parte de un módulo cargable del kernel, que puede cargarse y descargarse bajo demanda.

3.2.2.2 Monitorización y asignación de recursos desde espacio de usuario

Gracias a las modificaciones realizadas en PMCTrack como parte de este TFM, es posible utilizar desde espacio de usuario las extensiones de monitorización de Intel CMT, y MBM, y de particionado de caché con Intel CAT. En esta sección describimos cómo utilizar este soporte.

Esencialmente, el módulo de monitorización modificado, que hace uso del API de kernel presentada en la subsección anterior, exporta 3 contadores virtuales (`llc_usage`, `total_llc_bw`y `local_llc_bw`), que permiten consultar el uso de caché y el ancho de banda total y local, respectivamente.

Al cargar el módulo del kernel de PMCTrack, este módulo de monitorización (MM) no está activo típicamente. Podemos consultar qué MM está activo leyendo del fichero `/proc/pmc/mm_manager` (módulo activo marcado con `[*]`):

```
$ cat /proc/pmc/mm_manager
[*] 0 - This is just a proof of concept
[ ] 1 - IPC sampling SF estimation module
[ ] 2 - PMCtrack module that supports Intel CMT
[ ] 3 - PMCtrack module that supports Intel RAPL
```

Para activarlo, escribiremos la cadena “activate <ID>” en`/proc/pmc/mm_manager`, donde <ID> es el identificador del MM en esta plataforma (2 en este caso). A continuación comprobaremos que en efecto está activo dicho MM:

```
$ echo activate 2 > /proc/pmc/mm_manager
$ cat /proc/pmc/mm_manager
[ ] 0 - This is just a proof of concept
[ ] 1 - IPC sampling SF estimation module
[*] 2 - PMCtrack module that supports Intel CMT
[ ] 3 - PMCtrack module that supports Intel RAPL
```

Los contadores virtuales descritos anteriormente se listan con el siguiente comando

```
$ pmc-events -V
[Virtual counters]
llc_usage
```

```
total_llc_bw
local_llc_bw
```

Estos contadores virtuales se pueden usar en una sesión de monitorización iniciada desde espacio de usuario, con cualquiera de los componentes de PMCTrack descrito anteriormente. Por ejemplo, el siguiente comando, que hace uso de la herramienta de línea de comandos `pmctrack` permite iniciar una sesión de monitorización del programa `libquantum` donde se registran cada medio segundo (`-T 0.5`) las instrucciones retiradas (contador hardware convencional), el espacio usado en caché y el ancho de banda total con memoria. Los dos últimos eventos, cuyo valor se reporta en bytes, se configuran con la opción `-V` al tratarse de contadores virtuales:

```
$ pmctrack -c instr -E -V llc_usage,total_llc_bw -T 0.5 ./libquantum06
[Event-to-counter mappings]
pmc0=instr
virt0=llc_usage
virt1=total_llc_bw
[Event counts]
nsample    pid      event          pmc0         virt0         virt1
      1  21997       tick    2216084332      19726336    7424210301
      2  21997       tick    3153487404      20381696    7488471040
      3  21997       tick    3025540737      20611072    7521959936
      4  21997       tick    3275834489      20905984    6838681600
      5  21997       tick    3144971370      20676608    7789248512
      6  21997       tick    3083216989      20807680    7255195648
      7  21997       tick    3336307124      20742144    6628966400
      8  21997       tick    3061355142      20905984    7891386368
      9  21997       tick    3153299622      20971520    7061635072
     10  21997       tick    3336722045      20676608    6673137664
     11  21997       tick    3059010855      20611072    7951253504
     12  21997       tick    3153179702      20152320    6960185344
     13  21997       tick    3413568400      19431424    6769213440
     14  21997       tick    3067065077      20643840    7782760448
     . . .
```

Utilizando el comando auxiliar `pmc-metric` con una tubería, como se muestra a continuación, es posible transformar los datos de uso de LLC y ancho de banda a MBs y MBps respectivamente:

```
$ pmctrack -c instr -E -V llc_usage,total_llc_bw -T 0.5 ./libquantum06 | \
pmc-metric -m 'LLC_usage_MB=virt0/(1000**2)'  -m 'Bandwidth_MBps=virt1/etime_us'
nsample    pid      event LLC_usage_MB Bandwidth_MBps
      1  22054       tick    18.968750 12561.229857
      2  22054       tick    19.812500 12571.687446
      3  22054       tick    19.812500 12825.906101
      4  22054       tick    19.843750 11536.850934
      5  22054       tick    19.750000 13283.725834
      6  22054       tick    19.062500 12360.463717
      7  22054       tick    19.187500 11304.386237
```

```
 8  22054       tick    19.718750 13452.416000
 9  22054       tick    19.843750 12049.047533
10  22054       tick    20.000000 11308.557566
11  22054       tick    19.937500 13516.216792
12  22054       tick    19.906250 11827.351100
13  22054       tick    19.281250 11479.449683
14  22054       tick    19.843750 13249.522366
. . .
```

Para consultar el valor de las CBMs por cada clase de servicio (COS) o poder cambiar su valor debemos acceder al fichero especial `/proc/pmc/config` –fichero general de parámetros de configuración de PMCTrack. Al leer de dicho fichero se muestra el valor de cada uno de los registros asociado a cada COS (llc_cbm)[4]:

```
$ cat /proc/pmc/config
sched_sampling_period = 1000
kernel_buffer_size = 3952 bytes (26 samples)
rmid_alloc_policy=0 (FIFO)
cat_nr_cos_available=16
cat_cbm_length=20
llc_cbm0=0xfffff
llc_cbm1=0xfffff
llc_cbm2=0xfffff
llc_cbm3=0xfffff
llc_cbm4=0xfffff
llc_cbm5=0xfffff
llc_cbm6=0xfffff
llc_cbm7=0xfffff
llc_cbm8=0xfffff
llc_cbm9=0xfffff
llc_cbm10=0xfffff
llc_cbm11=0xfffff
llc_cbm12=0xfffff
llc_cbm13=0xfffff
llc_cbm14=0xfffff
llc_cbm15=0xfffff
```

La información mostrada por el comando anterior revela que la LLC tiene 20 vías (una vía por cada bit en la CBM) en el sistema, y que éste soporta 16 clases de servicio (hasta 16 particiones diferentes de la LLC). Con la configuración por defecto, que es la mostrada anteriormente, todas las particiones de caché tienen todas las vías asignadas.

Para modificar el valor de una CBM basta escribir la cadena "llc_cbm<K> <VAL_CBM>" en `/proc/pmc/config`, donde <K> es el número de COS y <VAL_CBM> el nuevo valor que deseamos asignar a la CBM. Por ejemplo, el siguiente

[4]A pesar de que el API de kernel desarrollada para Intel RDT soporta la gestión de multiples LLCs, el módulo de monitorización desarrollado solo funciona en sistemas de un solo socket (una LLC), como la plataforma experimental utilizada en este TFM.

comando crearía una partición de 4 vías asociada a COS 1, que usa las vías 0-3 de la LLC:

```
$ echo llc_cbm1 0xf >  /proc/pmc/config
```

Para poder asignar una aplicación a un COS específico en tiempo de lanzamiento, se desarrolló el script `set_cos` que se usa del siguiente modo:

```
set_cos <COS_ID> <COMANDO_A_EJECUTAR> [ARGUMENTOS]
```

El comando lanza el programa especificado por el comando, que puede aceptar argumentos, y asigna el número de COS indicado por <COS_ID>. `set_cos` es un script BASH que establece el COS pasado como argumento para el proceso actual (escribiendo en `/proc/pmc/config`) y a continuación invoca `exec` para cambiar el programa ejecutado por el proceso al que indica el usuario:

```bash
#!/bin/bash
cos_id=${1-1}
shift
echo "cos_id=${cos_id}" > /proc/pmc/config
exec $*
```

El módulo monitorización de PMCTrack para Intel RDT procesa la cadena "cos_id=<numero>" y asigna el valor especificado al proceso actual (`current`).

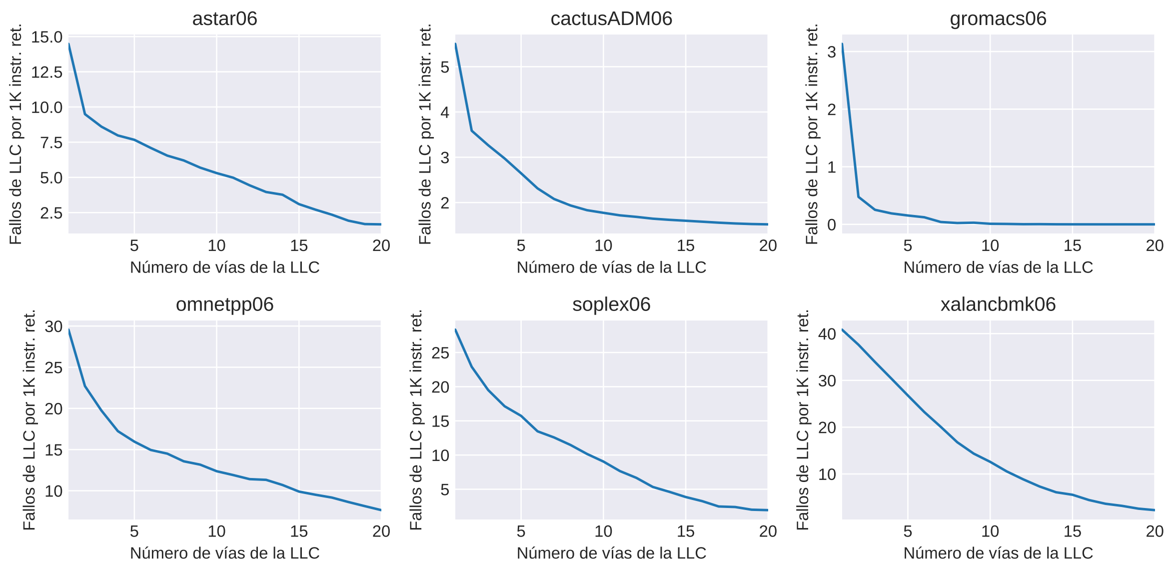


Figura 3.4: Curvas de fallos de LLC por cada 1000 instrucciones retiradas para distintas aplicaciones de SPEC CPU2006 obtenidas en nuestra plataforma experimental.

Empleando el soporte desarrollado en PMCTrack ha sido posible recabar toda la información de monitorización necesaria para las distintas aplicaciones que requiere

como entrada el simulador presentado en el capítulo 4. A modo de ejemplo, la figura 3.4 muestra las curvas de fallos de LLC por cada 1000 instrucciones retiradas (*Miss Rate Curves* o MRCs) obtenidos con PMCTrack para distintas aplicaciones de SPEC CPU2006. Estas curvas se han obtenido midiendo la tasa de fallos de caché media para la ejecución completa de las aplicaciones con distintos tamaños de caché (realizando un barrido de 1 a 20 vías de la LLC). La figura 3.5 muestra la degradación en rendimiento (*slowdown*) que cada aplicación experimenta al variar el tamaño de caché, con respecto a usar todo el espacio disponible en la LLC. El simulador que se presenta en el capítulo 4 hace uso de esta información para estimar el grado de rendimiento y justicia que ofrecen distintas estrategias de particionado de caché.

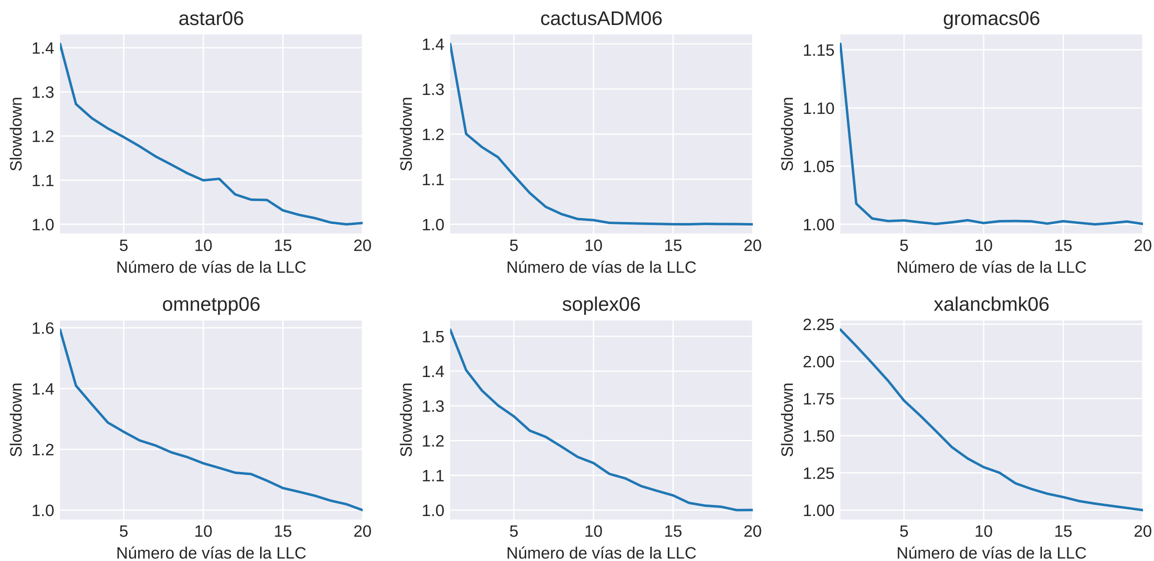


Figura 3.5: Degradación en el rendimiento (*slowdown*) observada al variar el tamaño de la LLC (número de vías) para distintas aplicaciones de SPEC CPU2006 ejecutándose en nuestra plataforma experimental.

Capítulo 4

Diseño e implementación del simulador de particionado de caché

En este capítulo se proporciona inicialmente una visión general del diseño del simulador, describiendo la estructura de los datos de entrada, la técnica utilizada para el cálculo aproximado del *slowdown* de cada aplicación de la carga de trabajo (en base al número de vías asignadas a cada aplicación de acuerdo al algoritmo de particionado seleccionado), así como la degradación estimada del rendimiento debido a la contención producida por el consumo de ancho de banda. Finalmente, se analizan detalles de implementación del simulador, describiendo el algoritmo paralelo de ramificación y poda (*Branch and Bound* - B&B) para la obtención de la solución óptima del problema del particionado de caché, así como las estrategias de poda diseñadas para los objetivos de optimización de la justicia y el rendimiento.

4.1 Diseño del simulador

4.1.1 Estructura de datos de entrada

El simulador propuesto tiene como objetivo no sólo el servir como herramienta para comparar fácilmente la eficacia de los distintos algoritmos de particionado de caché, sino también facilitar el diseño de nuevas técnicas de particionado. Para ello, es crucial que el simulador sea capaz de obtener de la forma más eficiente posible la solución óptima para distintos objetivos de optimización.

El simulador acepta como entrada dos ficheros en texto plano (CSV), uno que contiene la información de distintas métricas y otro que contiene la información de las cargas de trabajo multiprogramadas. El fichero de métricas contiene una tabla donde en cada fila se almacenan los distintos valores de métricas obtenidos en tiempo de ejecución

(instrucciones por ciclo, tasas de fallos de distintos niveles de la caché, uso de ancho de banda con la memoria principal, etc.) gracias a PMCTrack (usando los contadores hardware del procesador) para una determinada aplicación que se ejecuta sola en el sistema con un determinado número de vías asignados. Básicamente este fichero almacena el comportamiento de cada aplicación ejecutándose sola en el sistema para cada posible número de vías asignadas. Esta información se puede recopilar fácilmente en sistemas Intel modernos que cuentan con la tecnología Intel CAT. El archivo de cargas de trabajo contiene las diferentes cargas (una por línea) que serán consideradas por el simulador. Cada carga se codifica en el fichero como una secuencia de nombres de aplicaciones separadas por comas.

En el proceso de creación del archivo de métricas, el usuario puede decidir si incluir la información de la ejecución completa de una aplicación específica o solamente de una determinada fase de la ejecución de dicha aplicación. El *slowdown* de cada aplicación de cada carga para un determinado algoritmo de particionado se calcula a partir de la información proporcionada en este archivo. En la siguiente subsección se proporcionan más detalles sobre esto. A fecha de hoy, el simulador no permite almacenar ni procesar múltiples fases de ejecución de la misma aplicación de una determinada carga de trabajo. Se ha optado por realizar esta simplificación para que sea viable el cálculo de la solución óptima (haciendo uso de aplicaciones cuyas métricas permanecen estables para un determinado número de vías asignadas), de tal manera que de forma eficiente se pueda predecir el comportamiento de distintos algoritmos de particionado en cuanto a justicia y rendimiento. De este modo, el proceso de extracción de los valores de *STP* y *Unfairness* de una carga de trabajo para todos los algoritmos de particionado disponibles tan sólo lleva unos pocos segundos (excluyendo los algoritmos de búsqueda de la solución óptima). Nótese, que el hecho de que el simulador emplee los valores promedio de las métricas de ejecución, no impide que sea capaz de capturar las principales tendencias de los distintos algoritmos de particionado para cada carga de trabajo. Esto se analiza en detalle en la sección 6.2.

4.1.2 Cálculo del Slowdown

Por cada carga de trabajo y algoritmo de particionado especificado por el usuario en la línea de comandos, el simulador determina el valor estimado para las métricas *STP* y *Unfairness* (las cuales representan el rendimiento y el grado de justicia, respectivamente, tal y como se menciona en la sección 2.1).

El algoritmo de particionado en cuestión determina cómo se van a distribuir el total de vías de la caché entre las distintas aplicaciones de la carga de trabajo. Esta distribución de vías entre aplicaciones tiene un gran impacto en el *slowdown* de cada aplicación. Sin embargo, calcular el *slowdown* de cada aplicación basándose exclusivamente en su rendimiento ejecutándose sola en el sistema para distintos tamaños de caché (vías asignadas) puede proporcionar resultados poco realistas. Esto se debe a que las aplicaciones que

se ejecutan simultáneamente en el sistema también compiten por el uso de ancho de banda de la memoria principal [2]. Por tanto, para tener unos resultados más ajustados a la realidad, es crucial tener en cuenta la degradación del rendimiento que sufre cada aplicación debido a la contención del ancho de banda.

Sea A una carga de trabajo de n aplicaciones $[app_1, app_2, \cdots, app_n]$ ejecutándose en un sistema con una memoria caché de último nivel de k vías y bajo un algoritmo de particionado concreto. El algoritmo de particionado asignará a cada aplicación app_i un número de vías w_i (donde $\sum_{i=1}^{n} w_i = k$). El simulador estima el *slowdown* de cada aplicación app_i de la siguiente manera:

$$Slowdown_{app_i} = Slowdown_{cach,app_i} \cdot Slowdown_{bandwidth,app_i} \tag{4.1}$$

donde $Slowdown_{cach,app_i}$ indica la degradación en el rendimiento que sufre la aplicación app_i cuando tiene w_i vías asignadas con respecto a cuando tiene k (todas las vías de la LLC). $Slowdown_{bandwidth,app_i}$ indica el *slowdown* de la aplicación app_i debido exclusivamente a la contención en el consumo de ancho de banda, el cual tiene que compartirse con el resto de aplicaciones de la carga de trabajo.

Para el cálculo del $Slowdown_{cach,app_i}$, el simulador utiliza el número de instrucciones por ciclo (IPC) cuando la aplicación app_i usa k y w_i vías: $\frac{\text{IPC}(k)}{\text{IPC}(w_i)}$. Estos valores de IPC se proporcionan al simulador a través del archivo de métricas, descrito en la sección anterior. En cuanto al cálculo del $Slowdown_{bandwidth,app_i}$, se trata de una tarea más compleja principalmente por dos motivos. En primer lugar, porque el uso del ancho de banda de una determinada aplicación en ejecución depende del número de vías que tenga asignadas y del uso del ancho de banda de las demás aplicaciones ejecutándose simultáneamente en el sistema [18]. Normalmente, debido a la contención, el ancho de banda que se observa cuando la aplicación se ejecuta sola en el sistema con un determinado número de vías asignadas es mayor que cuando se ejecuta junto a otras aplicaciones de la carga de trabajo, lo cual trae consigo degradación del rendimiento. En segundo lugar, esta degradación del rendimiento fluctúa en función de lo sensible que sea cada aplicación a la contención del ancho de banda. En la siguiente sección del capítulo se describe el modelo de aproximación del ancho de banda que usa el simulador para calcular el $Slowdown_{bandwidth,app_i}$.

4.1.3 Modelo de contención del ancho de banda

Para tener en cuenta la contención del ancho de banda en el simulador, se usa el modelo probabilístico propuesto por Morad y otros [18]. A partir de datos de aplicaciones ejecutándose solas en el sistema, previamente recopilados, este modelo permite aproximar: (1) el ancho de banda que cada aplicación usaría cuando se ejecuta simultáneamente a

otras aplicaciones de la carga de trabajo y (2) el *slowdown* que proviene exclusivamente de la contención por uso del ancho de banda. Para determinar (1) en una carga de trabajo de n aplicaciones, se debe resolver el siguiente sistema de $n+1$ ecuaciones no lineales:

$$\left\{ \overline{B}_{s,i}^2 \cdot \left(1 - \frac{1}{\overline{B}_{a,i}}\right) + \overline{B}_{s,i} \cdot \left(1 - \frac{1}{\overline{T}}\right) \cdot \left(1 - \frac{1}{\overline{B}_{a,i}}\right) + 1 - \frac{1}{\overline{T}} = 0 \right\}_{i=1}^{n} \tag{4.2}$$

$$\sum_{i=1}^{n} \overline{B}_{s,i} = \overline{T} \tag{4.3}$$

donde n es el número de aplicaciones de la carga de trabajo, $\overline{B}_{a,i}$ es el ancho de banda usado por la aplicación i cuando se ejecuta sola en el sistema (con el mismo tamaño de caché disponible que cuando se ejecuta junto a las demás aplicaciones de la carga de trabajo), $\overline{B}_{s,i}$ es el ancho de banda usado por la aplicación i cuando se ejecuta junto al resto de aplicaciones de la carga de trabajo, y $\overline{T}$ es el uso total de ancho de banda consumido por la carga de trabajo. Cabe destacar que $\overline{B}_{a,i}$, $\overline{B}_{s,i}$ y $\overline{T}$ están normalizados con respecto al ancho de banda máximo teórico del sistema.

Para determinar el *slowdown* que proviene exclusivamente de la contención del ancho de banda (2), se usa el ratio $\overline{B}_{s,i}/\overline{B}_{a,i}$. Esta aproximación se basa en la idea de que el rendimiento de la aplicación disminuye proporcionalmente a la disminución del uso de ancho de banda, debido a la contención.

4.2 Implementación del simulador

4.2.1 Lenguaje y bibliotecas utilizadas en la implementación

Se ha decidido realizar la implementación del simulador usando el lenguaje Python, haciendo uso de sus bibliotecas disponibles para múltiples sistemas operativos. Esto permite que el simulador sea una herramienta multiplataforma. Se ha optado por usar este lenguaje debido a su facilidad de uso, y al hecho de que sea uno de los lenguajes de programación más usados hoy en día [31]. Debido a este uso masivo del lenguaje esperamos aumentar el número de usuarios que puedan colaborar añadiendo extensiones al simulador, como por ejemplo, incluyendo soporte de nuevas métricas de optimización o creando nuevos algoritmos de particionado de caché.

Para la implementación paralela del algoritmo de búsqueda de la solución óptima (la cual se verá en detalle más adelante en este mismo capítulo), que emplea una estrategia de B&B, se ha optado por usar el framework *ipyparallel* [32]. Este framework permite

realizar implementaciones paralelas, bien para la ejecución en una única máquina de memoria compartida o bien en un clúster. En el contexto del simulador, ya que la mayor parte de las tareas que debe realizar el algoritmo de búsqueda son completamente independientes entre sí, y es difícil predecir cuánto tardará en ejecutarse cada una, se ha optado por usar el modo *LoadBalance* de *ipyparallel*, el cual se basa en un planificador de tareas que de forma automática asigna cada tarea disponible a los distintos procesos *worker* –denominados *engines*–, a medida que éstos finalizan sus tareas.

Por otro lado, cabe destacar que, para resolver el sistema de ecuaciones no lineales requeridas para la evaluación del modelo de aproximación de la contención de ancho de banda descrito en la sección 4.1.3, se ha decidido usar la biblioteca *sympy*. Esta biblioteca ofrece un mecanismo flexible que permite especificar sistemas de ecuaciones no lineales usando código Python de alto nivel, resolviéndolos de forma eficiente. Adicionalmente, gracias a esta biblioteca es más fácil para otros usuarios ampliar el simulador creando y evaluando sus propios modelos de predicción basados en la métrica de *Slowdown*.

4.2.2 Búsqueda paralela de la solución óptima

El simulador hace uso de un algoritmo paralelo para determinar el particionado de caché óptimo para un determinado *target* de optimización. El algoritmo es el resultado de paralelizar el esquema de ramificación y poda (*Branch and Bound* - B&B) que se describe en el algoritmo 1.

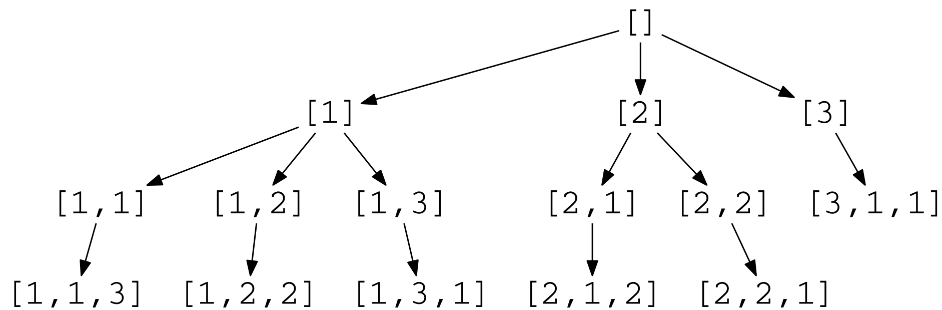


Figura 4.1: Árbol del espacio de búsqueda para el problema de particionado de caché óptimo con 3 aplicaciones y 5 vías.

Para realizar la paralelización de la búsqueda en el árbol, existen una serie de retos y problemas a los que tuvimos que enfrentarnos. En primer lugar, el espacio de búsqueda para el problema de particionado de caché óptimo es un árbol desequilibrado. Como ejemplo ilustrativo, en la figura 4.1 se muestra el árbol correspondiente para 3 aplicaciones y 5 vías. Los nodos hoja del árbol representan las posibles soluciones disponibles para el

Algoritmo 1: Función recursiva de ramificación y poda

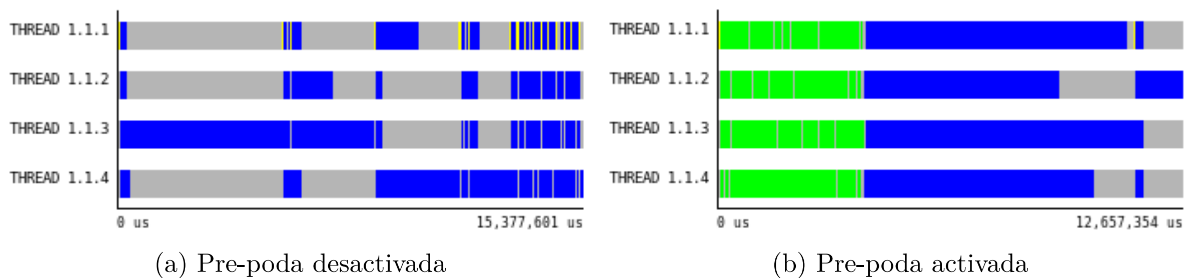
```
Input: workload es una lista de objetos aplicación, best_cost es el coste de la mejor
       solución encontrada hasta ahora (usada para la poda), nr_ways es el número
       de vías pendientes de asignar, y partial_sol mantiene la solución parcial para
       el nodo actual del árbol de búsqueda.
function get_optimal_partitioning(workload, best_cost, nr_ways, partial_sol):
    remaining_apps ← |workload| − |partial_sol|
    if remaining_apps = 1 then
        partial_sol ← Asigna nr_ways a la app restante
        total_cost ← Calcula el coste de partial_sol
        return (total_cost,partial_sol)
    end
    else if remaining_apps = nr_ways then
        partial_sol ← Asigna 1 way a cada app restante
        total_cost ← Calcula el coste de partial_sol
        return (total_cost,partial_sol)
    end
    else
        max_ways ← nr_ways − remaining_apps − 1
        for assigned_ways ← 1 to max_ways do
            subsol ← añade assigned_ways a partial_sol
            ideal_cost ← Coste estimado de una solución ideal alcanzable a partir de
             subsol
            if ideal_cost es mejor que best_cost then
                rem_ways ← nr_ways − assigned_ways
                (c, s) ←
                 get_optimal_partitioning(workload, best_cost, rem_ways, subsol)

                Actualizar (best_cost,best_sol) con (c, s) si c es mejor que best_cost
            end
        end
    end
    return (best_cost,best_sol)
```

número de vías y de aplicaciones mencionados anteriormente, mientras que los nodos intermedios representan soluciones parciales. Para simplificar, cada solución se representa como un vector donde el valor existente en la posición i-ésima, $i = \{0, 1, \cdots, n\}$, indica el número de vías asignadas a la aplicación $i + 1$. Por ejemplo, la solución asociada con el nodo de más a la izquierda del árbol en la figura 4.1 es $[1, 1, 3]$ lo que quiere decir que las dos primeras aplicaciones de la carga de trabajo obtienen una vía cada una mientras que la última obtiene tres.

Una forma eficaz de mejorar el rendimiento del algoritmo de B&B secuencial puro es dividir el árbol en subárboles cuyo nodo raíz se encuentra en un cierto nivel del árbol (por ejemplo, el nivel 2), procesar de forma paralela cada uno de los subárboles y finalmente obtener la solución final a partir de las soluciones de cada proceso paralelo.

La solución óptima se determinaría a partir de las mejores soluciones resultantes del procesamiento paralelo de cada uno de los subárboles. Desafortunadamente, como ya se ha comentado anteriormente, el árbol del espacio de búsqueda no está equilibrado, lo que da lugar a una carga de trabajo muy desigual entre los subprocesos paralelos y, en consecuencia, a una escalabilidad limitada. Para minimizar este problema, se ha dotado al simulador de un mecanismo que divide el árbol en subárboles con un tamaño similar, es decir, cada subárbol tiene un número similar de soluciones de acuerdo a la ecuación 2.4. Esto implica considerar, para el procesamiento paralelo, subárboles cuyos nodos raíz comienzan a diferentes niveles del árbol.



(a) Pre-poda desactivada (b) Pre-poda activada

Figura 4.2: Trazas del framework Paraver obtenidas para una carga de 6 aplicaciones ejecutándose de forma paralela en la plataforma B usando 4 cores. Las tareas de pre-poda están representadas en verde, las ”normales” de B&B en azul, y los periodos inactivos en gris.

Sin embargo, el procesamiento de cada subárbol de manera completamente independiente no permite alcanzar un buen rendimiento, ya que la solución ideal usada para la poda de todos estos subárboles es la de la solución inicial obtenida mediante un algoritmo heurístico[1]. Siguiendo esta aproximación, no es posible usar la mejor solución encontrada en un determinado subárbol para podar ramas de otros subárboles. Para mitigar este problema, el simulador emplea un procedimiento iterativo (véase el paso 3 en el algoritmo 2) en el que en cada iteración se procesan un determinado número de subárboles en paralelo, tantos como nodos de cómputo hayan sido configurados en el parámetro *chunk* del simulador. La mejor solución que se encuentra como resultado del procesamiento de esos subárboles en paralelo se usa para podar el siguiente grupo de subárboles en la siguiente iteración del bucle. El valor por defecto del parámetro *chunk* es 2, el cual proporciona los mejores resultados para la mayoría de las cargas de trabajo usadas en el capítulo 6.

Debido a la naturaleza impredecible de la poda, el procesamiento de subárboles de tamaño similar en distintos subprocesos sigue sin garantizar una carga uniforme entre los mismos. Se puede dar el caso de que un determinado subárbol se poda por completo al

[1]Para obtener la solución inicial, se utiliza una variante de UCP [7] que se alimenta con tablas de *slowdown* por aplicación y número de vías. Esta variante mejora claramente los resultados de la versión UCP original, que utiliza tablas MPKI.

comprobarse que la solución ideal (cota superior/inferior de la solución actual) alcanzable desde la raíz del subárbol es peor que la mejor solución obtenida hasta el momento. En este caso, el subproceso correspondiente queda inactivo, mientras otros pueden estar procesando subárboles llenos de soluciones prometedoras, lo que provoca un desequilibrio en la carga. Para ilustrar este problema se han generado trazas de ejecución que se han procesado de forma offline por la herramienta Paraver [33], la cual genera una representación gráfica de la ejecución de los diferentes cores[2]. En la figura 4.2a se ilustra el problema de desequilibrio comentado.

Para minimizar los efectos negativos de la poda en el contexto del equilibrio de la carga, el simulador cuenta con una fase previa a la fase paralela llamada fase de *pre-poda*. En esta fase se "podan" subárboles completos si se detecta que de éstos no se va a obtener ninguna solución prometedora. Esto se hace al comparar la solución inicial, obtenida a partir del algoritmo heurístico, con la mejor solución ideal alcanzable desde cada uno de los subárboles. La pre-poda nos permite reducir sustancialmente el número de subárboles que serán procesados en la fase paralela de B&B. En la figura 4.2b se ilustra el impacto de la pre-poda en el procesamiento de los subárboles (tareas en verde). En el ejemplo, la pre-poda permite acelerar la ejecución (4 cores) en un factor de 1,21X con respecto a la ejecución con la fase de pre-poda desactivada. Adicionalmente cabe destacar que, tal y como se puede ver en la figura, a diferencia del procesamiento real de los subárboles (tareas en azul), las tareas de pre-poda tienen una complejidad computacional muy similar, por lo que la fase de pre-poda realizada en paralelo goza de una alta escalabilidad.

En el algoritmo 2 se describe el algoritmo paralelo que utiliza el simulador. Básicamente consta de 3 pasos o fases: (1) dividir el árbol de espacio de búsqueda en subárboles con un tamaño similar, esta fase es la única de las tres que es secuencial; (2) aplicar la pre-poda en cada uno de los subárboles obtenidos en la fase anterior, para quedarnos sólamente con aquellos subárboles prometedores; y (3) buscar la solución óptima a partir del procesamiento paralelo de los subárboles prometedores. Cabe destacar que la efectividad de las fases paralelas (2 y 3) depende enormemente del valor del parámetro *high_threshold* utilizado en la fase 1. Un valor muy alto de este parámetro suele provocar que la pre-poda sea ineficaz, ya que se suelen podar menos subárboles cuanto más grandes son éstos. Por el contrario, si asignamos un valor muy bajo, aumenta el número de subárboles, y con ello la complejidad computacional asociada a la pre-poda, lo que a su vez puede generar una sobrecarga considerable. Realizando un estudio de sensibilidad hemos seleccionado el valor 200 como el valor por defecto de este parámetro, ya que es el que mejor rendimiento ha dado para la gran mayoría de las cargas de trabajo exploradas en nuestro análisis experimental (capítulo 6). La principal conclusión que obtenemos del estudio de sensibilidad es que la mejor elección de los parámetros depende en gran medida de la efectividad de la función de poda, que a su vez depende

[2]Adrián García, colaborador externo de dirección de este TFM, ha instrumentado el código del simulador para la extracción de trazas.

Algoritmo 2: Función paralela de ramificación y poda.

Input: *workload* es una lista de objetos aplicación, *nr_ways* es el número de vías disponibles para asignar, *prepruning* permite activar o desactivar la fase de pre-poda, y *chunk* es el límite de subárboles que pueden ser procesados por core.

```
function get_optimal_solution_parallel(workload, nr_ways, high_threshold,):
// Fase 1: División del árbol en subárboles
    (default_cost, def_sol) ← Aplica la estrategia UCP al workload usando las
      métricas de Slowdown
    partial_sols ← Divide el espacio de búsqueda total en subárboles de tamaño
      similar, asegurándose de que cada partial_sol no supera el límite de
      high_threshold soluciones a explorar
// Fase 2: Pre-poda paralela
    if prepruning then
        parallel foreach partial_sol do
            subsol_cost ← Estima el mejor coste posible a partir de partial_sol
            if subsol_cost < default_cost then eliminarlas de partial_sols
        end
    end
// Fase 3: Procesamiento paralelo de los subárboles prometedores
    while |partial_sols| > 0 do
        sols_to_explore ← Selecciona nr_cores * chunk subsoluciones de
          partial_sols
        parallel foreach core do
            core_sols ← Selecciona chunk subsoluciones de sols_to_explore
            (c, s) ←
              get_optimal_partitioning(workload, nr_ways, subsol_cost, core_sol)
            Actualizar (best_cost,best_sol) con (c, s) si c es mejor que best_cost
        end
    end
    return (best_cost,best_sol)
```

de cada carga de trabajo.

4.2.3 Función de poda

Uno de los aspectos imprescindibles para que el rendimiento del algoritmo de B&B sea el adecuado es la efectividad de la función de poda para un determinado *target* de optimización. Para mantener una implementación lo más flexible posible, la función del simulador que calcula el coste de una determinada solución para un *target* de optimización acepta como parámetro una solución parcial o completa. Si se proporciona una solución parcial (junto con las vías pendientes de asignar), la función automáticamente calcula una cota inferior (para métricas donde es mejor cuanto menor sea su valor) o superior (para métricas donde es mejor cuanto mayor sea su valor) para cualquier solución

alcanzable a partir de esa solución parcial, distribuyendo las vías restantes entre el resto de aplicaciones. A continuación se explica cómo se construye esta cota inferior/superior para las métricas de *Unfairness* y *STP* presentadas en la sección 2.1, usadas para medir el grado de justicia y rendimiento de los diferentes algoritmos de particionado de caché.

El cálculo de ambas cotas se basa en la observación de que el *slowdown* de una aplicación disminuye a medida que asignamos más vías a dicha aplicación. Debido a que las métricas comentadas se definen en términos del *slowdown*, esta observación es crucial para determinar el valor de las cotas. Para el *STP* calculamos la cota superior asociada (al ser una métrica donde es mejor cuanto mayor es el valor) a partir de la solución ideal que resulta de asignar la mayor cantidad de vías posibles a aquellas aplicaciones que aún no tienen vías asignadas en la solución parcial. Sea R el número de vías restantes a asignar y S el número de aplicaciones que se consideran para repartir las vías restantes. En cualquier posible distribución de R vías entre las S aplicaciones, cada aplicación podrá tener como máximo $R - S + 1$ vías. Por tanto, la cota superior de la métrica *STP* dada una solución parcial es el valor de *STP* de la solución ideal que resulta de completar la solución parcial con la asignación de $R - S + 1$ vías a las aplicaciones restantes. Por ejemplo, considérese un sistema con una memoria caché de 10 vías y una carga de trabajo de 4 aplicaciones. Para la solución parcial [3, 2], la solución ideal para la métrica *STP* sería [3, 2, 4, 4]. Obviamente, esta solución ideal no coincide con ninguna solución factible, ya que la suma de las vías asignadas a cada aplicación supera el total de vías disponibles en el sistema. Sin embargo, este cálculo de la cota superior para la métrica *STP* da lugar a una poda muy efectiva.

La solución ideal que acabamos de ver para la métrica de *STP* no tiene porqué coincidir con la solución ideal para la métrica de *Unfairness* (la cual es mejor cuanto menor sea su valor). Tal como se puede ver en la ecuación 2.2, el valor de esta métrica es el cociente entre el *slowdown* máximo y mínimo observados entre todas las aplicaciones de la carga de trabajo. Asignar a una aplicación restante $R - S + 1$ vías puede hacer que su *slowdown* supere el máximo hasta ahora, o éste sea inferior al mínimo, haciendo que, en cualquier caso, el valor de la métrica aumente. Por tanto, la cota inferior de la métrica *Unfairness* dada una solución parcial es el valor de *Unfairness* considerando sólo las aplicaciones de la solución parcial (tal como hemos visto, incluir más aplicaciones sólo puede hacer que el valor de *Unfairness* aumente).

Capítulo 5

Guía de usuario básica del simulador

En este capítulo se proporcionan las instrucciones básicas para utilizar el simulador desde el punto de vista del usuario. En primer lugar se describen los pasos para instalarlo, enumerando las dependencias y configuraciones necesarias. Finalmente se describen las opciones más importantes con un ejemplo de uso.

5.1 Instalación del simulador

La instalación del simulador se puede llevar a cabo en cualquier plataforma con el entorno Python instalado. Sin embargo, esta guía se centra en la instalación en sistemas tipo UNIX (GNU/Linux o MacOSX). Para la instalación del simulador es necesario instalar, además del entorno Python, las siguientes dependencias (todas ellas se encuentran incluidas en los repositorios por defecto del gestor de paquetes de Python *pip*):

- matplotlib
- pandas
- numpy
- sympy
- ipyparallel
- sklearn
- datetime
- pytz

Tras la instalación de *ipyparallel*, es probable que los comandos encargados del arranque del clúster no estén incluidos en el PATH del sistema[1]. Para añadirlos, en primer lugar

[1] Para más información sobre ipyparallel, consúltese la documentación oficial en https://ipyparallel.readthedocs.io/en/latest

es necesario buscar el lugar donde se encuentran instalados, lo cual puede hacerse con el siguiente comando:

```
$ find ~ -name 'ipcluster'
```

Una vez obtenida la ruta al directorio de los binarios, lo añadimos al PATH usando el siguiente comando:

```
$ export PATH=$PATH:<RUTA IPYPARALLEL>
```

5.2 Uso básico del simulador

Para usar el simulador es necesario realizar unos pequeños pasos previos. En primer lugar, es preciso acceder al directorio raíz del simulador.

```
$ cd <DIRECTORIO RAÍZ SIMULADOR>
```

Desde este directorio, establecemos las variables de entorno necesarias por el simulador para que pueda ser ejecutado, con el siguiente comando.

```
$ . shrc
```

A continuación, arrancamos un clúster local de 8 cores con *ipyparallel*.

```
$ ipcluster start -n 8 --daemonize
```

Finalmente, ejecutamos el simulador, usando el lanzador *./test/sim.py*, como en el siguiente ejemplo.

```
./test/sim.py -s data/ms.csv data/ws.csv -a opt-stp,ucp -f table -b 35000 -p -r 1
W#   Algorithm   BenchID   Name          Mask/NR_WAYS   STP       Slowdown
W1   opt-stp     1         wupwise00     0x1(1)         0.97598   1.02462
W1   opt-stp     2         zeusmp06      0x2(1)         0.96532   1.03593
W1   opt-stp     3         xalancbmk06   0x7fc(9)       0.98013   1.02027
W1   opt-stp     4         OVERALL       0.1527s        2.92142   1.01534
W1   ucp         1         wupwise00     0x1(1)         0.97598   1.02462
W1   ucp         2         zeusmp06      0x2(1)         0.96532   1.03593
W1   ucp         3         xalancbmk06   0x7fc(9)       0.98013   1.02027
W1   ucp         4         OVERALL       0.0011s        2.92142   1.01534
```

La interfaz de línea de comandos del simulador cuenta con numerosas opciones. En el ejemplo anterior se han usado las más comunes, que se describen a continuación.

- `-s` especifica la ruta al fichero de métricas. Dicho fichero contiene una tabla en la que en cada fila se almacenan los valores de distintas métricas para una determinada aplicación ejecutándose sola en el sistema con un número determinado

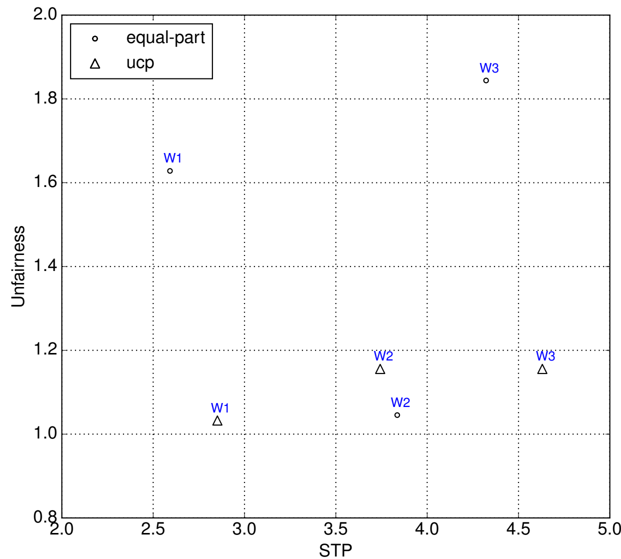


Figura 5.1: Gráfica generada por el simulador, usando el lanzador *./test/sim.py*, con el siguiente comando: `./test/sim.py -s data/metrics.csv data/ws.csv -a equal-part,ucp -f quiet -C`.

de vías asignadas. Dichos valores se obtienen de forma offline usando los contadores hardware del procesador, usando una herramienta como PMCTrack.

- El siguiente parámetro (obligatorio) especifica la ruta a un fichero de cargas de trabajo, el cual contiene las diferentes cargas (una por línea) que serán consideradas por el simulador. Cada carga de trabajo se codifica en el fichero como una lista de nombres de aplicaciones separados por comas.
- `-a` especifica un conjunto de algoritmos de particionado a simular, separados por comas.
- `-f` define el formato en que se mostrará la información. Existen numerosos formatos que se pueden usar, los más importantes son los siguientes:
  - `full`: Muestra la información en formato log. Es el formato definido por defecto.
  - `table`: Muestra la información en formato de tabla. Es el formato utilizado en el ejemplo anterior.
  - `cluster`: Muestra la información en formato clúster (reducido).
  - `quiet`: No muestra ninguna información por la salida estándar.
- `-p` activa la ejecución paralela del algoritmo de ramificación y poda (*Branch and Bound* - B&B) para el cálculo del particionado óptimo.
- `-b` define el máximo ancho de banda en memoria (en MBps) disponible en la simulación.
- `-r` permite definir un rango de cargas de trabajo a utilizar en la simulación,

respecto al conjunto de cargas completo. Se define como una lista de elementos separados por comas. Cada elemento puede ser un número de cargas específico o un rango usando el carácter '-'. Por ejemplo, si se han especificado 10 cargas en el fichero y definida la opción `-r 1,3,5-7,9-` el simulador procesará las cargas 1, 3, 5, 6, 7, 9 y 10.

Para ver todas las opciones disponibles en el simulador, se puede usar la opción `-h`.

```
./test/sim.py -h
usage: sim.py [-h] [-s SUMMARY_FILE] [-b MAX_BANDWIDTH] [-p]
              [-H] [-C] [-f FORMAT] [-d] [-r USE_RANGE]
              [-O key=val] WORKLOAD_FILE

Test main file for simulator (UCP, Yu-Petrof, etc.)

positional arguments:
  WORKLOAD_FILE         a workload file

optional arguments:
  -h, --help            show this help message and exit
  -s SUMMARY_FILE, --summary-file SUMMARY_FILE
                        File with the data collected offline for every
                        benchmark
  -b MAX_BANDWIDTH, --max-bandwidth MAX_BANDWIDTH
                        Max bandwidth observed for the target machine
  -p, --parallel        Enable parallel search for the optimal
  -C, --generate-chart  Enable the STP-vs-UNF chart generator
  -f FORMAT, --format FORMAT
                        Format style for output
  -d, --debugging       Enable debugging mode (assume normal input files) and
                        keep workload 0
  -r USE_RANGE, --use-range USE_RANGE
                        Pick selected workloads only by specifying a range
  -O key=val, --option key=val
                        Use generic algorithm-specific options
```

Cabe destacar la opción `-C, --generate-chart`, que permite generar de forma automática una gráfica que enfrenta los valores de rendimiento y justicia para cada carga de trabajo y algoritmo de particionado especificado. Gracias a esta gráfica es posible comparar de un vistazo las características de los distintos algoritmos de particionado especificados. La figura 5.1 muestra un ejemplo de gráfica generada por el simulador.

Capítulo 6

Análisis experimental

Este capítulo contiene el análisis experimental realizado para poner a prueba distintos aspectos del simulador y del soporte de particionado de caché implementado en PMCTrack. Inicialmente se describe el entorno sobre el que se han llevado a cabo los experimentos, que emplea dos plataformas hardware distintas. Posteriormente se detallan los experimentos llevados a cabo para la validación del simulador, comparando los resultados obtenidos por el simulador con los de la máquina real. A continuación, se analiza la efectividad de la poda en el algoritmo de ramificación y poda (B&B) secuencial para la búsqueda de la solución óptima, evaluando el impacto que tiene la poda en el rendimiento. Finalmente, se analiza la escalabilidad del algoritmo de B&B paralelo.

6.1 Entorno experimental

Para poner a prueba diferentes aspectos del simulador, se han llevado a cabo experimentos en dos plataformas distintas con GNU/Linux, usando la versión 4.9.116 del kernel de Linux. La tabla 6.1 resume las características de las dos plataformas hardware utilizadas. Ambas plataformas usan procesadores Intel Xeon, aunque de distintas familias. La plataforma A está equipada con la tecnología *Intel Cache Allocation Technology*, que se ha utilizado para extraer los datos *offline* de rendimiento de distintos benchmarks, y que son

Plataforma	**A**	**B**
Modelo del procesador	Xeon E5-2620 v4	2 x Xeon E5-2695 v3
Frec. del procesador	2.1 GHz	2.3 GHz
Número de cores	8	28
Caché LLC (L3)	20MB/20 vías	35MB/20 vías
Memoria principal	32GB@2133 MHz	64GB@1600 MHz

Tabla 6.1: Características de las dos plataformas utilizadas

necesarios para el simulador. Para ello se ha usado el soporte para *Intel RDT* desarrollado en este TFM sobre PMCTrack, que se describe en el capítulo 3. Adicionalmente, esta plataforma A integra un procesador con microarquitectura Broadwell, una caché LLC inclusiva, y una granularidad de particionado de caché más fina que la de la plataforma B, que permite crear particiones de un mínimo de 1MB (20 vías).

La plataforma A se ha utilizado para llevar a cabo la validación del simulador (sección 6.2), donde se compara la información proporcionada por el simulador con la obtenida en la plataforma real para diferentes cargas de trabajo con diferentes algoritmos de particionado. La plataforma B, que cuenta con un mayor número de cores (28) que la A, se emplea para evaluar la efectividad de la poda en las versiones secuenciales y paralelas del algoritmo de B&B (sección 6.3) y para medir la escalabilidad del algoritmo paralelo (sección 6.4).

6.2 Validación del simulador

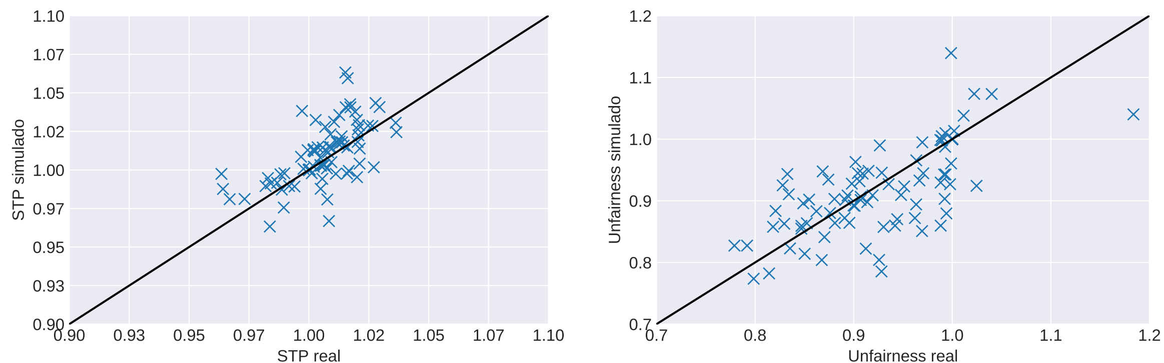


Figura 6.1: Comparativa entre los valores reales y los proporcionados por el simulador para las métricas de STP y Unfairness en la plataforma A, normalizados a los resultados de la estrategia de particionado Equal-Part.

Para llevar a cabo los experimentos de validación se han utilizado un total de 25 cargas de trabajo generadas de forma aleatoria, constando cada una de 6 a 8 aplicaciones. Para construir las cargas, se emplearon un total de 28 benchmarks diferentes de SPEC CPU. Para cada carga de trabajo obtuvimos los valores de rendimiento global y justicia proporcionados por el simulador para distintos algoritmos de particionado: UCP [7], Yu-Petrov [13], Equal-Part, y las soluciones de particionado óptimas calculadas por el simulador para las métricas de *STP* y *Unfairness*, denominadas *Opt-STP* y *Opt-Unf* respectivamente. Para validar estos resultados, se compararon con los observados en la máquina real aplicando las mismas estrategias de particionado de forma estática (misma para toda la ejecución), usando la misma máquina donde se obtuvo la información de rendimiento que se usó como entrada al simulador (Plataforma A).

Para los experimentos en la máquina real, se utilizó la herramienta PMCTrack [15] con el soporte para Intel RDT (véase el capítulo 3), que permite establecer particiones de caché por proceso desde el espacio de usuario en sistemas equipados con Intel CAT. Esencialmente, antes de ejecutar la carga de trabajo en la máquina real, se ejecuta en el simulador con un determinado algoritmo de particionado para obtener el particionado de la caché para ese algoritmo en cuestión, y se asigna cada partición a la aplicación correspondiente de la carga de trabajo. Por simplicidad, el particionado de caché es el mismo (estático) durante toda la ejecución de la carga de trabajo. Todas las aplicaciones de la carga de trabajo se lanzan simultáneamente, y cuando una de ellas termina, ésta se reinicia en repetidas ocasiones hasta que la aplicación más larga de la carga termina tres veces. Posteriormente se mide el rendimiento y la justicia, utilizando la media geométrica de los tiempos de ejecución de cada aplicación.

La figura 6.1 muestra la comparativa entre los valores reales y los proporcionados por el simulador para las métricas de *STP* y *Unfairness* en la plataforma A. Ambas métricas se han normalizado a los resultados de la estrategia de particionado Equal-Part. Como se puede observar, los valores de *STP* y *Unfairness* proporcionados por el simulador se aproximan de forma significativa a los valores reales observados. La tasa de error promedio observada es del 1% para *STP* y del 3% para *Unfairness*. La razón principal de las simulaciones menos precisas tiene que ver con las imprecisiones asociadas al modelo de contención de ancho de banda para aplicaciones que utilizan mucho ancho de banda con memoria. En particular, estas imprecisiones se deben a que el simulador utiliza el ancho de banda promedio resultante de la ejecución completa para predecir el *slowdown* derivado de la contención. En ocasiones, no representa el comportamiento de determinadas fases de la aplicación donde se encuentran picos en el consumo de ancho de banda que son muy superiores al promedio. En estos casos, el simulador tiende a predecir a la baja el *slowdown* debido a la contención surgida por la compartición del ancho de banda entre aplicaciones, lo que provoca que las estimaciones de los valores de la métrica de *Unfairness* sean inferiores a los valores reales. Este problema podría subsanarse haciendo que el simulador sea consciente de las distintas fases de ejecución de las distintas aplicaciones de la carga de trabajo. Sin embargo, esto haría aún más difícil determinar la solución óptima, ya que el particionado óptimo debería reevaluarse para cada fase. No obstante, cabe destacar que el objetivo principal del simulador es facilitar el diseño de nuevos algoritmos de particionado ligeros (baja sobrecarga) que puedan implementarse en el sistema operativo. Estos algoritmos podrían invocarse periódicamente, para poder reaccionar a los cambios de fase de las aplicaciones, estableciendo de forma dinámica particionados de caché de acuerdo al comportamiento de cada aplicación en tiempo real.

La figura 6.2 muestra la precisión de las predicciones del simulador para algunas de las cargas de trabajo consideradas. Los resultados en este caso se desglosan por cada estrategia de particionado. Como se puede observar, el simulador es capaz de capturar el beneficio relativo de una estrategia de particionado sobre los demás, en términos de justicia y rendimiento, y nos permite identificar las estrategias de particionado

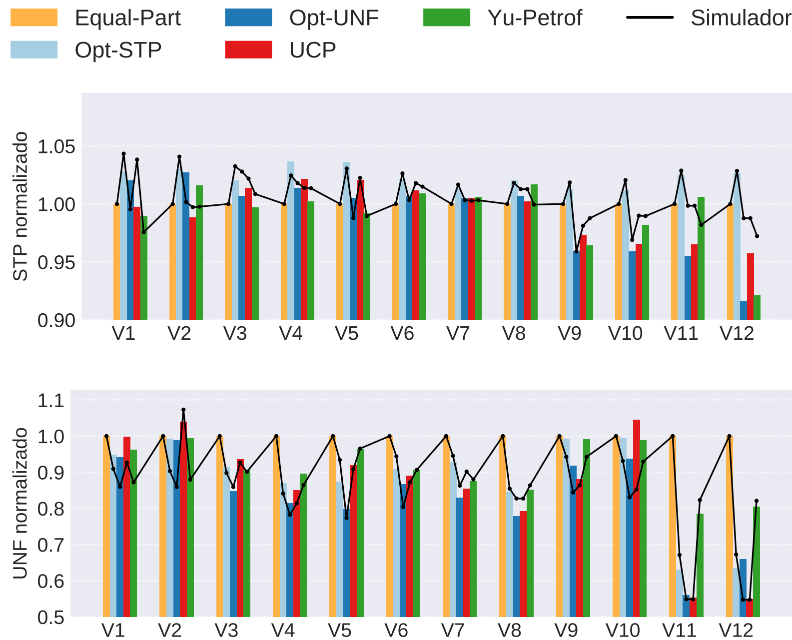


Figura 6.2: Comparativa entre los valores reales y los proporcionados por el simulador para las métricas de STP y Unfairness en la plataforma A, normalizados a los resultados de la estrategia de particionado Equal-Part.

que mejores resultados obtienen ante distintos escenarios. Los resultados revelan que dividir la LLC en particiones del mismo tamaño (Equal-Part) no proporciona buenos resultados en cuanto a justicia y rendimiento, ya que no se tiene en cuenta el grado de sensibilidad de cada aplicación a la memoria caché. El algoritmo de Yu-Petrov proporciona mejores resultados en algunos casos, pero también está sujeto a una alta degradación del rendimiento y la justicia. Finalmente, observamos que el algoritmo de UCP es más efectivo en general, pero en muchos casos aún está lejos de las soluciones óptimas Opt-STP y Opt-Unf. Cabe destacar que UCP depende de la capacidad de generar tablas MPKI en tiempo real. En la práctica, sin las extensiones hardware especiales [7] que aún no se han adoptado en procesadores comerciales, se requiere una monitorización potencialmente pesada en tiempo de ejecución, para obtener los datos para diferentes tamaños de caché y actualizarlos periódicamente para reflejar el comportamiento real de la aplicación a lo largo del tiempo. Un interesante uso futuro es usarlo para guiar el proceso de diseño de algoritmos de particionado de baja sobrecarga, más adecuados para su adopción en un sistema operativo real.

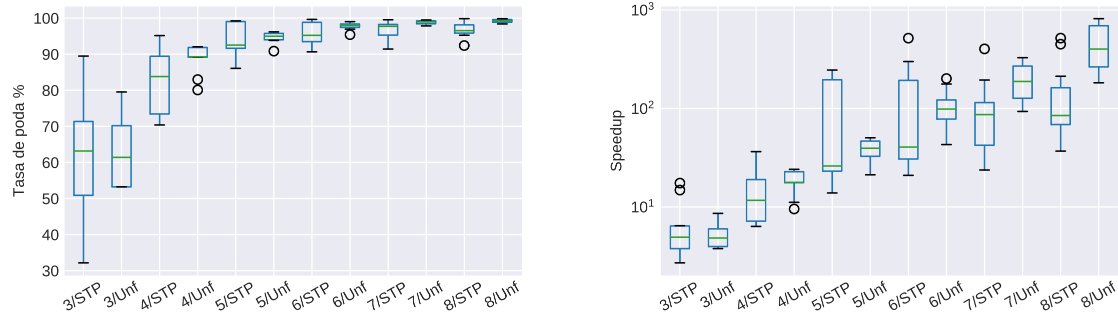


Figura 6.3: Tasa de poda (izquierda) y speedup (derecha) obtenidos para los diferentes conjuntos de cargas de trabajo. La etiqueta del eje X, con formato n/objetivo, indica el número de aplicaciones de la carga de trabajo (n) y el objetivo de optimización –rendimiento (STP) o justicia (Unf)–.

6.3 Efectividad de la poda

Una de las características más efectivas del algoritmo de B&B es la estrategia de poda para las métricas *STP* y *Unfairness* descrita en la sección 4.2.3. Para evaluar el impacto que tiene la poda en el rendimiento, se han creado de forma aleatoria 66 cargas de trabajo constando cada una de un conjunto de entre 3 y 8 aplicaciones (habiendo como máximo tantas aplicaciones ejecutándose como cores tiene la plataforma A). Se han considerado 11 cargas de trabajo por cada número de aplicaciones considerado (11 cargas de trabajo de 3 aplicaciones cada una, otras 11 de 4 aplicaciones, etc.). Se han descartado las cargas de trabajo de 2 aplicaciones debido al bajo número de soluciones posibles (tantas como el número de vías de la caché disponibles menos una). Para estos experimentos, se ha alimentado al simulador con los datos experimentales recabados *offline* para la plataforma A.

La figura 6.3 muestra la *tasa de poda* y el *speedup* para los diferentes conjuntos de cargas de trabajo (clasificados por número de aplicaciones de la carga) y objetivos de optimización (rendimiento y justicia) obtenidos de la ejecución de la versión secuencial del algoritmo de B&B. El *speedup* se ha calculado con respecto a la ejecución secuencial del simulador con la poda desactivada. La tasa de poda se define como el porcentaje sobre el número total de soluciones posibles (nodos hoja del árbol de búsqueda completo) que se descartaron por poda en el algoritmo de B&B. Hay que tener en cuenta que esta tasa se ha obtenido utilizando el número de soluciones procesadas por el algoritmo de búsqueda: nodos hoja del árbol más soluciones parciales (nodos intermedios) donde el algoritmo descarta el procesamiento del subárbol correspondiente debido a la poda.

Los resultados revelan que la tasa de poda depende en gran medida de la naturaleza de la carga de trabajo. Por ejemplo, considerando las cargas de trabajo de 3 aplicaciones para el objetivo de optimización del rendimiento, la tasa de poda varía entre el 32,2%

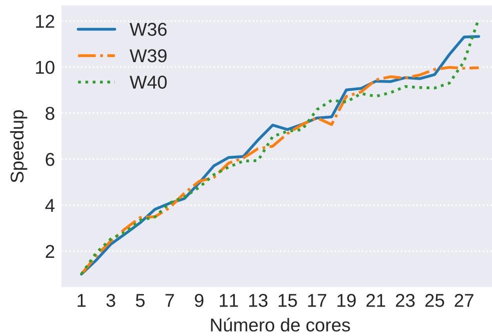
y el 89,5%. También se observa que, a medida que consideramos cargas de trabajo de más aplicaciones, la variabilidad comentada anteriormente disminuye y la tasa de poda promedio mejora de forma significativa; para cargas de trabajo de más de 5 aplicaciones la tasa de poda supera el 94%. Esto indica que la efectividad de la poda aumenta a medida que aumenta el tamaño del problema, siendo ésta una muy buena propiedad del simulador.

Los resultados del *speedup* (gráfica con escala logarítmica) revelan que la poda acelera la ejecución del simulador en un factor de hasta 815,8X (observado para una carga de trabajo de 8 aplicaciones). Además, tal y como se esperaba, la tasa de poda tiene un impacto muy significativo en el *speedup*. Por ejemplo, la pequeña superioridad que tiene la función de poda para la métrica de *Unfairness* respecto a la de *STP* para cargas de 6-8 aplicaciones da lugar a *speedups* sustancialmente mayores. Y lo que es más importante, puesto que el tamaño del espacio de búsqueda crece exponencialmente con el número de aplicaciones, un pequeño aumento en la tasa de poda puede tener un gran impacto en el *speedup*; para cargas de trabajo de 8 aplicaciones, un aumento del 0,5% de la tasa de poda puede acelerar la ejecución en un factor de 2X.

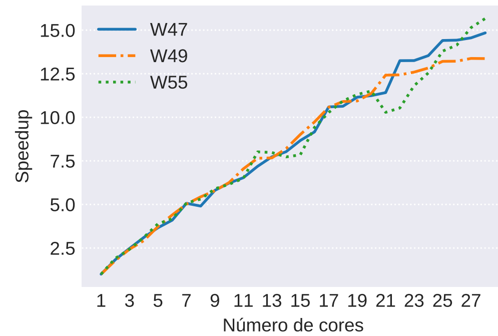
6.4 Análisis de la escalabilidad

Para evaluar la escalabilidad del simulador para la versión paralela del algoritmo de B&B para la búsqueda del particionado óptimo, se ha utilizado la plataforma B, que cuenta con 28 cores distribuidos en dos sockets (véase la tabla 6.1 para más detalles). Para el análisis, se ha utilizado un subconjunto de las cargas de trabajo utilizadas en la sección anterior, donde la ejecución secuencial del simulador, con la poda habilitada, tardaba más de un minuto en encontrar la solución que optimiza el rendimiento (métrica *STP*). En los experimentos, se han utilizado estos valores para los parámetros del simulador: $high_threshold = 200$ y $chunk = 2$. Estos valores se han fijado en base a un estudio de sensibilidad, el cual ha determinado que esos valores son los que proporcionan el mejor rendimiento para la mayoría de las cargas de trabajo. La principal conclusión que se saca de este estudio de sensibilidad es que el mejor valor de estos parámetros depende en gran medida de la efectividad de la poda, que a su vez depende de la naturaleza de las aplicaciones (tal y como se vió en la sección anterior).

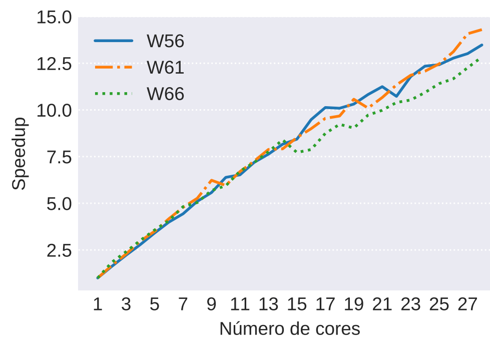
La figura 6.4 muestra el *speedup* para cada conjunto de cargas de trabajo (constando cada uno de 3 cargas de trabajo de 6, 7 y 8 aplicaciones, respectivamente), variando el número de cores disponibles de 1 a 28. Cabe destacar que el *speedup* se ha calculado con respecto al uso de un solo core con la poda habilitada, por lo que este factor de aceleración complementa el que proporciona la poda. Los resultados revelan que, en general, el *speedup* aumenta de forma lineal con el aumento del número de cores. El mayor factor de aceleración alcanzado es 15,7X (para la carga de trabajo 55 con 28 cores).



(a) 6 aplicaciones



(b) 7 aplicaciones



(c) 8 aplicaciones

Figura 6.4: Muestra de la escalabilidad para diferentes cargas de trabajo de 6, 7 y 8 aplicaciones.

A pesar de que el *speedup* alcanzado aún está lejos del valor ideal (28), la tendencia lineal que revelan las gráficas indican una buena escalabilidad. Esta tendencia se ha logrado gracias a la fase de pre-poda del algoritmo paralelo de B&B, la cual es altamente escalable. En algunos casos, la fase de pre-poda es tan efectiva que no deja suficiente trabajo para mantener todos los cores ocupados en la última etapa del algoritmo. En la figura 6.5 se puede observar un ejemplo de este comportamiento, donde se muestran las trazas de la ejecución del algoritmo para la carga de trabajo 51 en la plataforma B. En este caso particular, podríamos mitigar el problema de desequilibrio calculando el número de soluciones posibles de cada tarea B&B (número de nodos hoja del subárbol) y partiendo aquellas tareas con un gran número de soluciones en tareas más pequeñas, con idea de distribuir más trabajo entre los cores. Aunque este enfoque puede mejorar ligeramente el *speedup*, esto no soluciona completamente el problema de desequilibrio, ya que la complejidad computacional asociada al procesamiento de tada tarea B&B depende en gran medida de la efetividad impredecible de la poda en el subárbol correspondiente. Dado que este problema no supone un impedimento para que el simulador escale linealmente con el incremento de cores, se ha dejado como trabajo futuro la realización

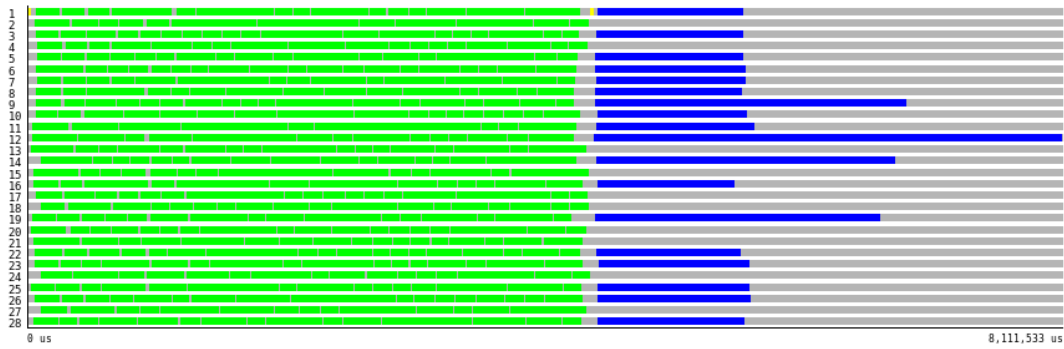


Figura 6.5: Trazas de ejecución para la carga de trabajo 51 en la plataforma B, con 28 cores.

de un estudio más a fondo sobre este problema de desequilibrio.

Capítulo 7

Conclusiones y trabajo futuro

En este capítulo se resumen las conclusiones finales de este Trabajo Fin de Máster y se proponen líneas de trabajo futuro que han surgido tras la realización del mismo.

7.1 Conclusiones

En este TFM se ha incorporado el soporte necesario en la herramienta PMCTrack para explotar la funcionalidad proporcionada por la tecnología *Intel Resource Director* (RDT) que está presente en procesadores actuales de gama alta de Intel (p.ej., familia Xeon). Esta tecnología permite, entre otras cosas, que el sistema operativo (SO), hipervisor o monitor de máquina virtual (VMM) distribuya de forma flexible el espacio disponible en el último nivel de caché (*Last-Level Cache* - LLC) entre distintas aplicaciones o máquinas virtuales que se ejecutan simultáneamente. Para ello el hardware expone al software de sistema una serie de registros que permiten realizar particionado de caché por vías (*way partitioning*).

El soporte de particionado de caché asistido por hardware supone una gran noticia para el usuario final, ya que el software de sistema puede potencialmente explotar técnicas de particionado de la LLC de forma transparente a las aplicaciones para mitigar los problemas asociados a la contención por la compartición de la LLC, como es el caso de la degradación del rendimiento global, de la justicia o incluso de la eficiencia energética de la plataforma [28]. No obstante, particionar la caché de forma óptima para un determinado objetivo resulta una tarea muy compleja [8].

Para afrontar este reto, se ha desarrollado, también como parte de este TFM, un simulador de particionado de caché que permite encontrar la solución óptima de particionado para distintos objetivos de optimización. El simulador utiliza un algoritmo paralelo de ramificación y poda (*Branch and Bound* - B&B) para la búsqueda eficiente de dicha solución óptima. El principal objetivo que ha motivado el desarrollo de este simulador es

ayudar a los desarrolladores de sistemas operativos y VMMs a diseñar nuevas estrategias de particionado con baja sobrecarga que puedan adoptarse en sistemas reales. Gracias al simulador desarrollado, es posible realizar rápidamente una comparativa entre distintos algoritmos de particionado de caché y la solución óptima para distintos *targets* de optimización. Esto permite determinar si una estrategia de particionado específica es o no prometedora, incluso antes de proceder a su implementación y evaluación en un sistema operativo real. A la vista de los experimentos de validación del simulador, llevados a cabo en una plataforma con soporte de Intel RDT, podemos concluir que el simulador es capaz de identificar con éxito el mejor algoritmo de particionado de caché para un conjunto de cargas de trabajo.

Un aspecto clave del simulador es que tiene en cuenta el *slowdown* que sufre cada aplicación al competir con otras por el acceso a dos recursos compartidos clave de los sistemas multinúcleo actuales: la caché de último nivel (LLC) y el ancho de banda con memoria principal. Para aproximar la degradación del rendimiento por la contención derivada de la compartición del ancho de banda con memoria, se ha implementado en el simulador el modelo probabilísitico propuesto en [18]. Para evaluar este modelo es necesario resolver un sistema de ecuaciones no lineales, lo cual tiene asociado un elevado coste computacional, especialmente para el cálculo del particionado de caché óptimo, donde el modelo de ancho de banda se tiene que aplicar en reiteradas ocasiones. Afortunadamente, gracias a la efectividad de las estrategias de poda, diseñadas específicamente para el algoritmo de B&B del simulador, junto a la buena escalabilidad que ofrece este algoritmo, la solución óptima se puede obtener en un tiempo razonable (tiempos no superiores a un minuto para las cargas de trabajo utilizadas en nuestros experimentos).

Para alimentar al simulador con datos de rendimiento de las aplicaciones en una plataforma concreta– lo cual es necesario para aproximar el comportamiento de cualquier estrategia de particionado de caché en una máquina real–, es preciso extraer una serie de métricas de rendimiento de distintas aplicaciones para distintos tamaños de caché (diferente número de vías) mediante los contadores hardware del procesador. Para ello empleamos el soporte de particionado de caché desde espacio de usuario que forma parte de las extensiones de la herramienta PMCTrack desarrolladas a lo largo de este TFM.

Todo el código desarrollado en este TFM, tanto las extensiones de la herramienta PMCTrack como el simulador de particionado de caché, es *open-source* (licencia GPL) y se encuentra ya disponible en sus respectivos repositorios de *GitHub* [16], [34].

7.2 Trabajo futuro

Como continuación natural de este TFM, se pueden considerar dos principales líneas de trabajo futuro. Por una parte, tal y como se comentó en la sección 2.3, debido a la gruesa granularidad del particionado de caché que permiten los procesadores modernos equipados con la tecnología *Intel CAT*, el particionado de caché estricto puede resultar

ineficaz en ciertas ocasiones [14]. Para hacer frente a este problema, se podría dotar al simulador de soporte para el particionado de caché en clúster, donde se contempla la compartición de vías entre aplicaciones. Esto permitiría potencialmente mejorar el rendimiento gracias al aumento de la granularidad, al posibilitar que dos o más aplicaciones compartan una misma partición de la LLC. El principal reto que surge al intentar ampliar la funcionalidad el simulador con soporte para análisis de estrategias de particionado de caché en clúster, es cómo aproximar la fracción de espacio en caché (y el correspondiente *slowdown*) que dos o más aplicaciones obtienen cuando se asignan a la misma partición (*cluster*). Como aproximación inicial para realizar esta predicción podría emplearse el modelo simple propuesto por Mukkara y otros [22] .

Otra vía principal de trabajo futuro es la realización de un estudio más a fondo acerca del problema de desequilibrio en el algoritmo de ramificación y poda (B&B) paralelo, que surge por la naturaleza impredecible de la efectividad de la poda. Este problema se mitiga en muchas simulaciones gracias a la inclusión de la fase de pre-poda, que evita el procesamiento de subárboles que no tienen soluciones prometedoras. Lamentablemente la pre-poda no consigue solucionar el problema completamente, tal y como se discutió en el capítulo 6.

Apéndice A

Código fuente del proyecto

Todo el código desarrollado en este TFM es *open-source* (con licencia GPL) y está disponible en *GitHub*.

A.1 Soporte para Intel RDT en PMCTrack

El código asociado a la inclusión del soporte para Intel RDT en PMCTrack se encuentra en el repositorio oficial de PMCTrack:

- *https://github.com/jcsaezal/pmctrack*

Y más concretamente, los ficheros modificados son los siguientes:

- *https://github.com/jcsaezal/pmctrack/blob/master/src/modules/pmcs/intel_cmt_mm.c*
- *https://github.com/jcsaezal/pmctrack/blob/master/src/modules/pmcs/monitoring_mod.c*
- *https://github.com/jcsaezal/pmctrack/blob/master/src/modules/pmcs/include/pmc/intel_cmt.h*

A.2 Simulador de particionado de caché

El código asociado al simulador de particionado de caché se encuentra en el siguiente repositorio:

- *https://github.com/pbbcache/cachesim*

Apéndice B

Introduction

Chip multicore processors (CMPs) are today the predominant architecture in general purpose computing systems. Despite their benefits, these processors pose a number of challenges to system software. One of the main challenges is how to mitigate the negative effects that occurs when co-running applications compete for the usage of shared resources [1]. Note that the cores in a CMP system are not completely independent processing units, but typically share the last-level cache (LLC) and other memory resources, such as the DRAM controller [2], [3].

As an example, the figure B.1 shows the existence of a third level of cache (L3) shared between the different cores of the Intel Xeon 2620 E5-v4 processor, present in the experimental platform used in this work. As shown in the figure, the chip integrates a single DRAM controller, shared between the cores. Applications running simultaneously on this system may compete for space in the LLC or for the use of memory bandwidth, which can lead to substantial performance degradation, potentially very uneven between applications [2], [4]–[6].

Shared LLC partitioning –that is, assigning a cache partition of a certain size to each application of a workload– has proven effective in mitigating the effects of shared resource contention [7], [8]. Recently, Intel processors have been equipped with hardware support for cache partitioning, thanks to Intel Cache Allocation Technology (CAT) [9] - part of Intel Resource Director Technology (RDT) [10]-. Intel CAT allows the operating system (OS), hypervisor, or virtual machine monitor (VMM) to flexibly distribute the available space in the LLC among different applications or virtual machines running simultaneously. Similar cache partitioning support has also been adopted for some high-performance processors with ARM architecture, such as Cavium Thunder X [11], [12].

Numerous cache partitioning algorithms designed to optimize different target metrics [6]–[8], [13], [14] have been proposed to date. However, a thorough evaluation of these algorithms has not yet been performed to assess how close they come to the optimal

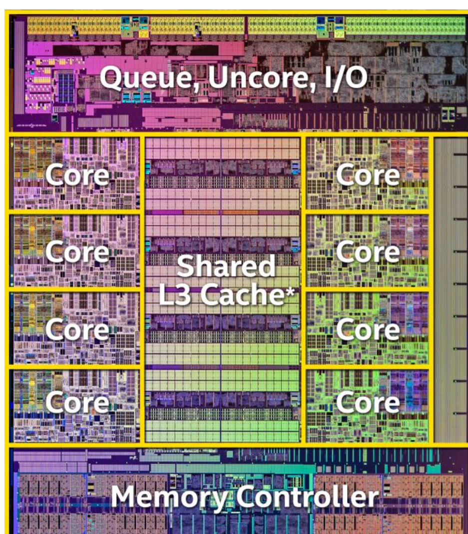


Figura B.1: Shared resources between cores in the Intel Xeon 2620 E5-v4 processor (*Broadwell-EP*)

solution for different optimization objectives. This makes it difficult to quantify their true potential. As an additional problem, these cache partitioning algorithms do not accurately take into account the contention derived from the competition for bandwidth, which depends largely on the partitioning applied [13]. This additional source of contention can substantially degrade the performance of each individual application, reducing the benefits of cache partitioning, making it sometimes even counterproductive [1], [2].

B.1 Project goals and work plan

The main objective of this Masters' Thesis has been the creation of a research framework that would allow to face the problems previously described. Specifically, the framework has been developed to guide the process of designing cache partitioning policies based on Intel RDT technology, and to allow the rapid evaluation of each policy's effectiveness. This framework consists of two components: (1) a set of PMCTrack [15] monitoring tool extensions specific to the use of Intel RDT on real hardware, and (2) a cache partitioning simulator.

Extensions created in PMCTrack provide support for cache partitioning and monitoring the bandwidth consumed by an application. These new features, which have already been added to the official repository of this *open-source* tool [16], can be used both from within the operating system (OS) –using a kernel-level API– and from user space. In this Masters' Thesis, intensive use has been made of features accessible from user space, to collect experimental performance data that the developed simulator requires to work. However, thanks to the OS support developed for Intel RDT it is possible to make

modifications to the Linux kernel to partition the LLC in a way that is transparent to users, and based on the characteristics of the applications, to optimize different objectives. Although the implementation of cache partitioning strategies in the OS is beyond the scope of this Masters' Thesis, research has already been done where the framework developed has been used for this purpose [17].

The cache partitioning simulator, also developed as part of this Masters' Thesis, has a dual function. First, it is a tool for prototyping cache partitioning algorithms that makes it possible to quickly determine (with little development effort) whether a particular algorithm is promising or not, from the point of view of overall performance, fairness or other relevant metrics. Second, the simulator efficiently determines the optimal cache partitioning for different scenarios and optimization objectives using a parallel branch and bound algorithm specifically designed to efficiently distribute the computation across multiple cores. The ability to compare the effectiveness of a particular partitioning algorithm with the optimal solution is key to assessing its true potential.

To determine the performance degradation (slowdown) of a multi-programmed workload –necessary to evaluate the effectiveness of different partitioning policies–, the simulator is equipped with a prediction model that takes into account the impact of contention on the LLC and that caused by competition between applications for the usage of memory bandwidth. To determine performance degradation due to contention in the LLC only, the simulator uses application performance information (e.g., instructions per cycle) collected offline with processor hardware counters for different cache sizes. To quantify the negative impact on performance associated with memory bandwidth contention, the simulator considers –in addition to the information obtained offline– the probabilistic model proposed in [18], whose evaluation requires the resolution of a system of non-linear equations.

In order to proceed with the development and evaluation of the framework, a work plan was established consisting of the following tasks:

- Study of Intel's official documentation on *Resource Director Technology* (RDT) [10], with special emphasis on *Cache Allocation Technology* (CAT) and *Memory Bandwidth Allocation* (MBA).
- Implementation of Intel RDT support in the PMCTrack tool.
- Design and implementation of the simulator based on the slowdown prediction model described above. In this phase the design of the API for the prototyping and evaluation of partitioning algorithms was also carried out.
- Implementation of the sequential version of the optimal search algorithm (branch and bound), and design of two pruning functions associated with the optimization of the overall performance and the degree of fairness of the system, respectively.
- Development of the parallel version of the simulator for the search of the optimal solution, distributing in an efficient way the computation between the cores and treating in an effective way the scenarios of load imbalance due to the unequal nature of the pruning in different areas of the search space.

- Experimental evaluation of the simulator, including a validation of the results provided by the simulator (comparing them with those obtained in real hardware), and a study of performance and scalability of the parallel algorithm for finding the optimal solution.

B.2 Masters' Thesis structure

The rest of this Masters' Thesis is structured as follows:

- **Chapter 2** introduces the metrics used to quantify the degree of overall performance and fairness of different partitioning policies. It also formally describes the problem of optimal cache partitioning (strict and in clusters) and discusses related work.
- **Chapter 3** describes the technologies covered by *Intel Resource Director Technology* (RDT). Subsequently, the general characteristics of PMCTrack are presented and details are provided on the implementation of the Intel RDT support for this tool.
- Aspects of design and implementation of the cache partitioning simulator are presented in **chapter 4**. This chapter also describes in detail the structure of the simulator input data, the technique used to approximate the slowdown of each application when sharing resources with others, the parallel algorithm for finding the optimal solution, and the pruning strategies designed for different optimization objectives.
- A basic tutorial on how to use the simulator, including installation instructions, is provided in **chapter 5**.
- **Chapter 6** contains the experimental analysis carried out with the simulator, which includes a series of experiments to validate the simulator and the analysis of different aspects of the algorithm to find the optimal solution.
- In **chapter 7**, the final conclusions of the work are presented and possible future lines of work are proposed.
- Finally, three appendices are provided. The first (A) indicates the location of the project code, and the other two include (B) Introduction and (C) Conclusions and future work, translated into English.

Apéndice C

Conclusions and future work

This chapter summarises the final conclusions of this Masters' Thesis and proposes future lines of work that have arisen after its completion.

C.1 Conclusions

In this Masters' Thesis, the necessary support has been incorporated into the PMCTrack tool to exploit the functionality provided by the *Intel Resource Director Technology* (RDT) that is present in current high-end Intel processors (e.g., Xeon family). This technology allows, among other things, the operating system (OS), hypervisor or virtual machine monitor (VMM) to flexibly distribute the available space of the last level cache (LLC) among different applications or virtual machines running simultaneously. For this purpose, the hardware exposes to the system software a series of registers that allow way partitioning.

Hardware-assisted cache partitioning support is good news for the end user, as system software can potentially exploit LLC partitioning techniques transparently to applications to mitigate problems associated with LLC sharing contention, such as degradation of overall performance, fairness, or even energy efficiency of the [28] platform. However, partitioning the cache optimally for a given target is a very complex task [8].

To address this challenge, a cache partitioning simulator has also been developed as part of this Masters' Thesis, which enables the optimal partitioning solution to be found for different optimization purposes. The simulator uses a parallel Branch and Bound (B&B) algorithm for the efficient search for such an optimal solution. The main objective that has motivated the development of this simulator is to help operating system and VMM developers to design new partitioning strategies with low overhead that can be adopted in real systems. Thanks to the developed simulator, it is possible to quickly make a comparison between different cache partitioning algorithms and the

optimal solution for different optimization targets. This makes it possible to determine whether or not a specific partitioning strategy is promising, even before proceeding with its implementation and evaluation in a real operating system. In view of the simulator validation experiments, carried out on a platform with Intel RDT support, we can conclude that the simulator is capable of successfully identifying the best cache partitioning algorithm for a set of workloads.

A key aspect of the simulator is that it takes into account the slowdown that each application suffers when competing with others for access to two key shared resources of the current multicore systems: the last level cache (LLC) and main memory bandwidth. In order to approximate the performance degradation due to memory bandwidth contention, the probabilistic model proposed in [18] has been implemented in the simulator. To evaluate this model it is necessary to solve a system of non-linear equations, which has associated a high computational cost, especially for the calculation of optimal cache partitioning, where the bandwidth model has to be applied repeatedly. Fortunately, thanks to the effectiveness of the pruning strategies, specifically designed for the B&B algorithm of the simulator, together with the good scalability offered by this algorithm, the optimal solution can be obtained in a reasonable time (times not exceeding one minute for the workloads used in our experiments).

To feed the simulator with application performance data on a particular platform –which is necessary to approximate the behavior of any cache partitioning strategy on a real machine–, it is necessary to extract a series of performance metrics from different applications for different cache sizes (different number of ways) using the processor's hardware counters. For this, we use the cache partitioning support from user space that is part of the PMCTrack tool extensions developed throughout this Masters' Thesis.

All the code developed in this Masters' Thesis, both the PMCTrack tool extensions and the cache partitioning simulator, is open-source (GPL license) and is already available in their respective *GitHub* repositories [16], [34].

C.2 Future work

As a natural continuation of this Masters' Thesis, two main lines of future work can be considered. On the one hand, as commented in sección 2.3, due to the coarse granularity of cache partitioning allowed by modern processors equipped with *Intel CAT* technology, strict cache partitioning can sometimes be ineffective [14]. To deal with this problem, the simulator could be provided with support for cluster cache partitioning, where the sharing of cache ways between applications is contemplated. This could potentially improve performance by increasing granularity by allowing two or more applications to share the same LLC partition. The main challenge that arises when trying to extend the functionality of the simulator with support for the analysis of cache-clustering strategies, is how to approximate the fraction of cache space (and the corresponding slowdown)

that two or more applications get when they are assigned to the same partition (cluster). The simple model proposed by Mukkara and others [22] could be used as an initial approximation to make this prediction.

Another major avenue for future work is the conduct of a more in-depth study of the problem of imbalance in the parallel Branch and Bound (B&B) algorithm, which arises from the unpredictable nature of pruning effectiveness. This problem is mitigated in many simulations thanks to the inclusion of the pre-pruning phase, which avoids the processing of sub-trees that do not have promising solutions. Unfortunately, pre-pruning does not completely solve the problem, as discussed in chapter 6.

Bibliografía

[1] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, y M. Prieto, «Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors», *ACM Comput. Surv.*, vol. 45, n.º 1, pp. 4:1-4:28, dic. 2012.

[2] E. Ebrahimi, C. J. Lee, O. Mutlu, y Y. N. Patt, «Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems», en *15th Int'l Conf. Architectural Support Programming Lang. and Oper. Syst. (ASPLOS 10)*, 2010, pp. 335-346.

[3] A. Garcia-Garcia, J. C. Saez, y M. Prieto-Matias, «Contention-Aware Fair Scheduling for Asymmetric Single-ISA Multicore Systems», *IEEE Transactions on Computers*, vol. 67, n.º 12, pp. 1703-1719, dic. 2018.

[4] J. C. Saez, J. I. Gomez, y M. Prieto, «Improving Priority Enforcement via Non-Work-Conserving Scheduling», en *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, 2008, pp. 99-106.

[5] S. Zhuravlev, S. Blagodurov, y A. Fedorova, «Addressing Cache Contention in Multicore Processors Via Scheduling», en *15th Int'l Conf. Architectural Support Programming Lang. and Oper. Syst. (ASPLOS 10)*, 2010, pp. 129-142.

[6] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, y M. E. Gómez, «Application Clustering Policies to Address System Fairness with Intel's Cache Allocation Technology», en *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 194-205.

[7] M. K. Qureshi y Y. N. Patt, «Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches», en *Proceedings of MICRO 06*, 2006, pp. 423-432.

[8] S. Mittal, «A Survey of Techniques for Cache Partitioning in Multicore Processors», *ACM Comput. Surv.*, vol. 50, n.º 2, pp. 27:1-27:39, may 2017.

[9] K. Nguyen, «Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family». https://software.intel.com/en-us/articles/introduction-to-cache-

allocation-technology, 2016.

[10] «Intel® 64 and IA-32 Architectures Developer's Manual, Vol. 3: System programming guide». 2018.

[11] C. Inc., «ThunderX family of workload optimized processors». http://cavium.com/pdfFiles/ThunderX PB p12 Rev1.pdf, 2013.

[12] X. Wang, S. Chen, J. Setter, y J. F. Martínez, «SWAP: Effective Fine-Grain Management of Shared Last-Level Caches with Minimum Hardware Support», en *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 121-132.

[13] C. Yu y P. Petrov, «Off-chip Memory Bandwidth Minimization Through Cache Partitioning for Multi-core Platforms», en *Proceedings of the 47th Design Automation Conference*, 2010, pp. 132-137.

[14] N. El-Sayed, A. Mukkara, P. Tsai, H. Kasture, X. Ma, y D. Sanchez, «KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores», en *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 104-117.

[15] J. C. Saez, A. Pousa, R. Rodriguez, F. Castro, y M. Prieto-Matias, «PMCTrack: Delivering Performance Monitoring Counter Support to the OS Scheduler», *The Computer Journal*, vol. 60, n.º 1, pp. 60-85, 2017.

[16] PMCTrack, «Source code repository at Github». https://github.com/jcsaezal/pmctrack, 2019.

[17] A. Garcia-Garcia, J. C. Saez, F. Castro, y M. Prieto-Matias, «LFOC: A Lightweight Fairness-Oriented Cache Clustering Policy for Commodity Multicores», en *ICPP '19: Proceedings of the 2008 48th International Conference on Parallel Processing*, 2019.

[18] T. Y. Morad, N. Shalev, I. Keidar, A. Kolodny, y U. C. Weiser, «EFS: Energy-Friendly Scheduler for memory bandwidth constrained systems», *Journal of Parallel and Distributed Computing*, vol. 95, pp. 3-14, 2016.

[19] J. A. Joao, M. A. Suleman, O. Mutlu, y Y. N. Patt, «Utility-based acceleration of multithreaded applications on asymmetric CMPs», en *40th Ann. Int'l Symp. Computer Architecture (ISCA 13)*, 2013, pp. 154-165.

[20] D. Xu, C. Wu, P.-C. Yew, J. Li, y Z. Wang, «Providing Fairness on Shared-memory Multiprocessors via Process Scheduling», en *Proc. ACM Int'l Conf. Measurement and Modeling Comp. Syst. (SIGMETRICS 12)*, 2012, pp. 295-306.

[21] S. Eyerman y L. Eeckhout, «System-Level Performance Metrics for Multiprogram Workloads», *IEEE Micro*, vol. 28, n.º 3, pp. 42-53, may 2008.

[22] A. Mukkara, N. Beckmann, y D. Sanchez, «Whirlpool: Improving Dynamic Cache

Management with Static Data Classification», en *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 113-127.

[23] B. Gendron y T. G. Crainic, «Parallel branch-and-branch algorithms: Survey and synthesis», *Operations research*, vol. 42, n.º 6, pp. 1042-1066, 1994.

[24] T. G. Crainic, B. Le Cun, y C. Roucairol, «Parallel branch-and-bound algorithms», *Parallel combinatorial optimization*, vol. 1, pp. 1-28, 2006.

[25] J. Gmys, M. Mezmaz, N. Melab, y D. Tuyttens, «A GPU-based Branch-and-Bound algorithm using Integer–Vector–Matrix data structure», *Parallel Computing*, vol. 59, pp. 119-139, 2016.

[26] «Intel® 64 and IA-32 Architectures Developer's Manual, Vol. 4: Model-specific registers». 2018.

[27] «Intel® 64 and IA-32 Architectures Developer's Manual, Vol. 2: Instruction set reference». 2018.

[28] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, y M. Prieto, «Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors», *ACM Comput. Surv.*, vol. 45, n.º 1, pp. 4:1-4:28, dic. 2012.

[29] J. C. Saez *et al.*, «An OS-Oriented Performance Monitoring Tool for Multicore Systems», en *Proc. of Euro-Par 2015: Parallel Processing Workshops*, 2015, pp. 697-709.

[30] J. Saez, «PMCTrack monitoring modules». https://pmctrack.dacya.ucm.es/getting-started/#monitoring-modules, 2016.

[31] S. Cass y P. Bulusu, «Interactive: The Top Programming Languages 2018». https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018, 2018.

[32] «IPyparallel. Using IPython for parallel computing». https://ipyparallel.readthedocs.io/, 2018.

[33] BSC, «Paraver: a flexible performance analysis tool». https://tools.bsc.es/paraver, 2018.

[34] CacheSim, «Source code repository at Github». https://github.com/pbbcache/cachesim, 2019.