Soporte de calidad de servicio en Linux para procesadores equipados con la tecnología Intel CAT

TRABAJO FIN DE GRADO

Francisco Burruezo Aranda

Dirigido por: Juan Carlos Sáez Alcaide y Fernando Castro Rodríguez

Grado en Ingeniería de Computadores

Facultad de Informática

Universidad Complutense de Madrid

2018

Soporte de calidad de servicio en Linux para procesadores equipados con la tecnología Intel CAT

Memoria de Trabajo Fin de Grado

Francisco Burruezo Aranda

Dirigido por: Juan Carlos Sáez Alcaide y Fernando Castro Rodríguez

Grado en Ingeniería de Computadores

Facultad de Informática

Universidad Complutense de Madrid

2018

"*No thief, however skillful, can rob one of knowledge, and that is why knowledge is the best and safest treasure to acquire.*"

- L. Frank Baum, The Lost Princess of Oz

Agradecimientos

En primer lugar, quiero agradecer a los directores de mi Trabajo de Fin de Grado, Juan Carlos Sáez Alcaide y Fernando Castro Rodríguez, todo el tiempo que me han dedicado para sacar este trabajo adelante además de todo el conocimiento que me han proporcionado. Lo guardaré como un tesoro.

No puedo tampoco olvidarme de aquellos compañeros con los que pasé horas, horas y horas trabajando para llegar hasta aquí. Vuestra capacidad de esfuerzo y vuestras ganas de querer alcanzar cualquier meta me sirvieron de inspiración. Gracias.

A mi familia, que me ha concedido el privilegio del tiempo. La oportunidad para hacer lo que siempre quise hacer. No lo hubiese conseguido sin vuestro apoyo. Nunca lo olvidaré.

Hace muchísimos años apareció en mi vida una persona que siempre confío en mí. Una persona que me levantaba cuando me caía. Una persona que siempre me dice, ¡tú puedes! Gracias por hacerme creer que todo es posible.

Resumen

Los procesadores multicore integran en un mismo chip múltiples núcleos de procesamiento; todos ellos comparten recursos, como niveles de la jerarquía cache o el controlador de memoria. Sin embargo, el hardware por sí mismo no otorga a las aplicaciones una fracción de los recursos compartidos proporcional a la prioridad que establece el usuario. Esto supone un serio problema para el sistema operativo, ya que la contención por recursos compartidos puede afectar muy negativamente a la calidad del servicio que el sistema ofrece.

Se propone en este trabajo un algoritmo de gestión de los recursos compartidos en tiempo real para optimizar la justicia durante la ejecución de procesos con la misma prioridad. Este algoritmo se implementa en un módulo del kernel Linux utilizando la herramienta open source de monitorización PMCTrack para hacer uso de su soporte de las tecnologías Intel CAT (Cache Allocation Technology) e Intel MBM (Intel Memory Bandwidth Monitoring), presentes en procesadores Intel de la familia Xeon E5-v4, así como para monitorizar métricas de rendimiento proporcionadas por los contadores hardware.

Palabras clave: Intel CAT, Intel MBM, Procesadores multicore, Algoritmo, Planificador, Recursos compartidos, Justicia, Kernel Linux, Contadores hardware.

Abstract

Multicore processors are made up multiple cores in the same chip; all of them shares resources, such as last levels in the cache hierarchy or the memory controller. Nevertheless, hardware doesn't provide by itself a proportional fraction of them in relation to user-defined task's priority. This is a a serious problem for the operating system, existing contention at shared resources can negatively affect to the quality of the service offered by the system to the user.

This work purposes a shared resources management real time algorithm in order to optimize fairness while are running tasks with the same priority. This algorithm is deployed on a Linux kernel module using the open source monitoring tool PMCTrack which supports Intel CAT (Cache Allocation Technology) and Intel MBM (Intel Memory Bandwidth Monitoring) technologies, integrated on Intel Xeon E5-v4 series, as well as the capability to monitor performance metrics through hardware counters.

Keywords: Intel CAT, Intel MBM, Multicore processors, Algorithm, Scheduler, Shared resources, Fairness, Linux kernel, Hardware counters.

Autorización de difusión y utilización

Los abajo firmantes, autorizan 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 Fin de Grado "Soporte de calidad de servicio en Linux para procesadores equipados con la tecnología Intel CAT", realizado durante el curso académico 2017-2018 bajo la dirección de Juan Carlos Sáez Alcaide y Fernando Castro Rodríguez 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.

Francisco Burruezo Aranda

Juan Carlos Sáez Alcaide

Fernando Castro Rodríguez

Índice general

Agradecimientos **iii**

Resumen **v**

Abstract **vii**

Autorización de difusión y utilización **ix**

1 Introducción **1**
1.1 Motivación 1
1.2 Objetivos del trabajo 2
1.3 Plan de trabajo 3
1.4 Organización de la memoria 4

2 Modelos de estimación de ancho de banda y de degradación del rendimiento **5**
2.1 Definición de slowdown 5
2.2 Definición de $BW_{slowdown}$ 6
2.2.1 Definición de BW_{alone} 6
2.2.2 Evaluación de la estimación de slowdown mediante el cálculo de $BW_{slowdown}$ 8
2.3 Definición de unfairness 11

3 Plataforma experimental y extensiones del hardware **13**
3.1 Características técnicas de la plataforma 13
3.2 Extensiones del hardware 14
3.2.1 Intel CAT - Cache Allocation Technology 14
3.2.2 Intel MBM - Memory Bandwidth Monitoring 15
3.3 Herramienta de monitorización PMCTrack 16

4 Diseño e implementación **19**
4.1 Diseño del módulo del kernel Linux 19
4.1.1 Información almacenada por aplicación 19
4.1.2 Diseño de algoritmos para reducir la métrica unfairness 21
4.1.3 Diseño del comportamiento del módulo del kernel Linux 22
4.2 Implementación del módulo del kernel Linux 25

4.2.1 Implementación de la interfaz de PMCTrack para módulos del kernel Linux . . . 25
4.2.2 Implementación de algoritmos en el módulo del kernel Linux . . . 27
4.2.3 Implementación de sistema de logs con ftrace . . . 31

5 Evaluación experimental **33**
5.1 Framework Het-Harness . . . 34
5.1.1 Configuración de Het-Harness . . . 34
5.1.2 Información proporcionada por Het-Harness . . . 38
5.2 Evaluación de la implementación . . . 41
5.2.1 Generador de cargas de trabajo . . . 41
5.2.2 Procesamiento de resultados . . . 42
5.2.3 Evaluación de los resultados . . . 44
5.3 Implementación de técnicas alternativas para la estimación de la degradación 49
5.3.1 Nueva definición de $BW_{slowdown}$ . . . 49
5.3.2 Estimación de coeficientes de regresión . . . 52
5.3.3 Implementación en el módulo del kernel Linux . . . 54
5.4 Resultados finales . . . 56
5.4.1 Minimizando la métrica unfairness . . . 56

6 Conclusiones y trabajo futuro **65**
6.1 Conclusiones . . . 65
6.2 Trabajo futuro . . . 66

Appendix A - Introduction **69**
Motivation . . . 69
Project goals . . . 70
Work plan . . . 71
Structure of the document . . . 71

Appendix B - Conclusions and future work **73**
Conclusions . . . 73
Future work . . . 74

Bibliografía **75**

Lista de tablas

3.1 Especificaciones de la plataforma experimental 13
3.2 Jerarquía de memoria cache del procesador Intel(R) Xeon(R) CPU E5-2620 v4 14

5.1 Resumen de la información ofrecida por el framework Het-Harness para una carga de trabajo 40
5.2 Resumen de la información ofrecida por el módulo del kernel Linux para una carga de trabajo 43
5.3 Estimación de BW_{alone} precisa 45
5.4 Estimación de BW_{alone} imprecisa 45
5.5 Cálculo de coeficientes de regresión mediante regresión 52
5.6 Cálculo de coeficientes de regresión mediante regresión aditiva 52
5.7 Configuración final del módulo del kernel Linux 57
5.8 Reducción de *unfairness* - Prueba de concepto 58
5.9 Reducción de *throughput* - Prueba de concepto 58
5.10 Reducción de *unfairness* en cargas de trabajo diversas 64
5.11 Reducción de *throughput* en cargas de trabajo diversas 64

Lista de figuras

2.1 Cálculo alternativo de BW_{alone} . . . 7
2.2 Evaluación positiva $BW_{slowdown}$ . . . 9
2.3 Evaluación negativa $BW_{slowdown}$ . . . 9
2.4 Ancho de banda requerido a lo largo del tiempo . . . 10

3.1 Intel CAT - Cache Allocation Technology . . . 15

5.1 Estimación BW_{alone} en entorno real . . . 45
5.2 Estimación *slowdown* en entorno real - Aplicaciones que maximizan la métrica *unfairness* . . . 46
5.3 Consumo de ancho de banda GemsFDTD - Independencia de la disponibilidad de cache . . . 47
5.4 Consumo de ancho de banda mcf - Uso irregular en función de disponibilidad de cache . . . 47
5.5 Consumo de ancho de banda bzip2 - Diferencia relevante de requisito de ancho de banda en función de disponibilidad de cache . . . 48
5.6 Estimación *slowdown* en entorno real - Aplicaciones con un consumo de ancho de banda independiente de la cache de último nivel que tengan disponible . . 48
5.7 Rectas de regresión resultantes de la corrección de la ecuación de $BW_{slowdown}$. 51
5.8 Reducción de *unfairness* en función de la carga de trabajo - Prueba de concepto 59
5.9 Grado de contención en los accesos a memoria principal insuficiente para aplicar los algoritmos encargados de reducir la métrica *unfairness* . . . 60
5.10 Ancho de banda consumido por aplicación y tiempos de desactivación 60
5.11 Carga de trabajo elevada donde se reduce la métrica *unfairness* . . . 61
5.12 Cargas de trabajo diversas donde se reduce la métrica *unfairness* . . . 63
5.13 Reducción de la métrica *unfairness* en contextos extremos . . . 63

Capítulo 1

Introducción

1.1 Motivación

Desde los años 80, la velocidad de los procesadores ha sido mejorada aproximadamente entre un 50% y un 100% cada año mientras que, en las memorias DRAM, sólo ha crecido alrededor de un 7% cada año [1]. El concepto 'memory wall' encuentra su significado en este contexto [2]; en el de la progresiva mejora en la velocidad de los procesadores en contraposición a los avances en la velocidad de estas memorias.

A principios de siglo, surgió la tendencia de aumentar el número de cores en un mismo procesador incrementando, aún más, la demanda de ancho de banda en memoria principal. Por ello, la importancia en el diseño de la jerarquía de memoria ha crecido en función de los avances en el rendimiento de los procesadores.

Están presentes hoy en día investigaciones centradas en aumentar el rendimiento que pueden ofrecer las memorias cache compartidas [3, 4] mientras que otros estudios giran en torno a disminuir la contención en la memoria principal. Esta problemática ha sido abordada desde puntos de vista distintos [5, 6, 7, 8, 9, 10, 11, 12]: desde planificar la ejecución de conjuntos de hilos en función de lo que interfieren unos con otros hasta asignar los recursos del sistema, entre todos los procesos en ejecución, teniendo en cuenta la justicia y el rendimiento.

Todas estas soluciones tienen un punto de encuentro en común: la necesidad de estimar la degradación que sufre una aplicación en el momento que esta compite con otras por los recursos compartidos; en este último contexto, por el acceso a la memoria principal.

Mientras que el ancho de banda que consume un proceso cuando se ejecuta simultáneamente con otros puede ser medido, el reto es realizar una estimación en tiempo real sobre el ancho de banda que consumiría la misma aplicación si esta se ejecutase sola en el mismo sistema. Los distintos patrones de acceso a memoria de las diferentes aplicaciones que se estén ejecutando al mismo tiempo en un sistema, la localización en memoria de los recursos solicitados por estas y la existencia de algoritmos de planificación de acceso a la memoria principal, contenidos en el controlador de memoria, hacen difícil realizar la estimación [7].

No se debe olvidar que el uso de los recursos compartidos cambia a lo largo del tiempo, al igual que los recursos disponibles [13], por lo que se antoja necesario poder realizar las estimaciones frecuentes y de forma eficaz.

En el contexto de la computación en la nube, donde a menudo se ejecutan aplicaciones muy diversas, resulta imprescindible garantizar calidad de servicio a todos los usuarios. El hardware, por sí mismo, no otorga a las aplicaciones una fracción de los recursos compartidos proporcional a la prioridad establecida para cada una de éstas. Puede presentarse el escenario en el que se ejecutan simultáneamente distintos procesos con la misma prioridad y que el rendimiento que ofrezcan estas aplicaciones sea muy distinto entre ellas [14].

Lo que se propone en este Trabajo de Fin de Grado es un algoritmo de gestión de recursos compartidos en tiempo real para optimizar la justicia [10, 14] durante la ejecución de procesos con la misma prioridad. La tecnología Intel CAT [15] permite mediante software poder gestionar el uso que hacen los procesos del último nivel de cache. En particular, el algoritmo está diseñado para realizar un reparto igualitario de este recurso y controlar el uso que hacen los procesos del ancho de banda con la memoria principal mediante desactivación puntual de cores.

Contando con la herramienta open source de monitorización PMCTrack [13, 16] y con el hardware disponible se pondrán a estudio diferentes técnicas [5, 7, 10, 9, 6] para realizar una planificación eficiente de tareas priorizando la justicia gestionando el uso del ancho de banda disponible.

1.2 Objetivos del trabajo

Para hacer posible una gestión justa de los recursos compartidos, en primer lugar fue necesario realizar un trabajo previo de investigación acerca de cómo poder estimar la degradación en el rendimiento que sufren los procesos en ejecución al tener que competir por el uso de éstos.

Se tomaron como base modelos y conclusiones obtenidas por otros autores [5] para estimar esta degradación en función de la degradación que sufren por tener que competir por los accesos a memoria principal. Por tanto, fue imprescindible validar cada una de ellas en la medida de lo posible. Esto pudo realizarse gracias a la herramienta open source PMCTrack[13, 16]. Concretamente, gracias a su interfaz de línea de comandos que ofrece información relativa a distintas métricas que puedan ser de interés para abordar cualquier problemática que se presente.

Además, PMCTrack [13, 16] proporciona soporte para monitorizar con un alto nivel de abstracción los contadores hardware presentes en la plataforma experimental, así como en cualquier otra compatible, en la implementación de módulos del kernel Linux. Contexto donde es posible desarrollar algoritmos que gestionen el uso de los recursos disponibles.

Después de hacer un estudio de distintas técnicas propuestas [5, 7, 10, 9, 6] relativas a la asignación de recursos compartidos se llevó a cabo la implementación de un algoritmo de gestión de los recursos compartidos en tiempo real en el kernel Linux. Este algoritmo hace uso

de las interfaces provistas por PMCTrack [13, 16] para monitorizar métricas proporcionadas por los contadores hardware y del soporte de las tecnologías Intel CAT [15] e Intel MBM [17]; utilizada esta última para supervisar el ancho de banda consumido por las aplicaciones que se encuentran en ejecución.

Contar con los benchmarks presentes en las suites SPEC CPU 2000 y SPEC CPU 2006 [18, 19] da la posibilidad de poner a examen las distintas técnicas estudiadas, y luego implementadas, con cargas de trabajo diversas.

Finalizada la implementación del algoritmo en un módulo del kernel Linux se llevó a cabo la fase experimental para verificar que las técnicas empleadas favorecen la justicia. Se hizo uso en esa etapa del framework Het-Harness que permite lanzar cargas de trabajo de manera sistemática bajo demanda.

1.3 Plan de trabajo

Para alcanzar estos objetivos se han llevado a cabo los siguientes pasos:

1. Utilización de la interfaz de línea de comandos de PMCTrack [13, 16].
2. Formación sobre el framework Het-Harness para llevar a cabo las fases experimentales.
3. Aprendizaje de Python como lenguaje de scripting para procesamiento de datos.
4. Estudio exhaustivo de distintas técnicas propuestas por otros autores [5, 7, 10, 9, 6] para el diseño de algoritmos. Elaboración, procesado e interpretación de experimentos asociados.
5. Formación acerca de la programación en el kernel Linux.
6. Utilización del software Weka [20] para el uso de algoritmos de machine learning.
7. Implementación del algoritmo de gestión de los recursos compartidos en un módulo del kernel Linux.
8. Evaluación de los resultados ofrecidos por la implementación.

El plan de trabajo expuesto es orientativo. Durante el desarrollo de este proyecto ha sido necesario profundizar en los aspectos internos de la herramienta PMCTrack [13, 16] con el paso del tiempo. Del mismo modo se ha hecho uso de distintos recursos escritos en C y en Python [21, 22, 23, 24] después de un paso previo de documentación, en función de las necesidades. Frecuente también ha sido la búsqueda de técnicas alternativas en el momento que aquellas, validadas en primera instancia y luego empleadas, no han resultado ser satisfactorias para el objetivo de este trabajo.

1.4 Organización de la memoria

La sucesión de capítulos que se presentan a continuación sigue el orden en el que se abordó este trabajo. La división de los contenidos está basada en las distintas fases que tuvieron lugar a lo largo del desarrollo del mismo. En cada uno de ellos se expone la motivación que supuso la realización de cada tarea, los pasos que se llevan a cabo para abordarla y las conclusiones que resultan relevantes.

Dado que es un trabajo que ha requerido llevar a cabo múltiples desarrollos, pequeños en su mayoría, se presentan también las secciones con mayor relevancia de estos. Caso excepcional es un capítulo dedicado exclusivamente al diseño y a la implementación del algoritmo de gestión de los recursos compartidos en tiempo real debido a su complejidad, extensión y transcendencia en este proyecto.

A continuación, se describe brevemente el contenido de los siguientes capítulos:

En el **capítulo 2** se hace un estudio acerca de la precisión de los modelos propuestos [5] que permiten estimar la degradación que sufren las aplicaciones al tener que competir por los recursos compartidos del sistema donde se ejecutan. Además, se definen conceptos recurrentes a lo largo del este documento.
En el **capítulo 3** se presenta la plataforma experimental, las extensiones hardware Intel CAT [15] e Intel MBM [17] y el soporte que proporciona PMCTrack [13, 16] para la realización de este trabajo.
En el **capítulo 4** se expone con alto nivel de detalle el diseño y la implementación del algoritmo de gestión de los recursos compartidos en tiempo real.
En el **capítulo 5** se realiza la evaluación de los resultados obtenidos haciendo uso del algoritmo propuesto.

Capítulo 2

Modelos de estimación de ancho de banda y de degradación del rendimiento

Este trabajo parte de realizar estimaciones precisas sobre la degradación que sufren las aplicaciones cuando compiten por los recursos compartidos del sistema donde se ejecutan. Por ello, ha sido imprescindible tener en consideración el trabajo previamente propuesto por diversos autores [5] para abordar esta tarea. Esta estimación resulta imprescindible para el propósito de este trabajo, favorecer la justicia [10, 14] durante la ejecución de procesos con la misma prioridad en el uso de los recursos del sistema.

En este capítulo se definirán conceptos que aparecerán de manera recurrente en el presente documento al mismo tiempo que se evaluarán las distintas proposiciones tomadas.

2.1 Definición de slowdown

El rendimiento, o *throughput*, de cualquier proceso que haga uso, en mayor o en menor medida, de la memoria principal es mermado por otras tareas que se encuentran compitiendo por este mismo recurso. Esta degradación en el rendimiento recibe el nombre de *slowdown*.

Los términos *slowdown* y *throughput* serán citados con frecuencia en el presente documento. Se definen tal que:

$slowdown = \frac{IPC_{alone}}{IPC_{shared}} = \frac{Tiempo_de_ejec_{shared}}{Tiempo_de_ejecuc_{alone}}$

$throughput = \sum_{i=0}^{n} \frac{1}{slowdown_i}$

Siendo IPC_{alone} el número de ejecuciones por ciclo de un proceso cuando se ejecuta solo en el sistema e IPC_{shared} cuando este se ejecuta simultáneamente con otras tareas en el mismo sistema.

Si tratamos de maximizar el *throughput* de un conjunto de tareas se nos presenta el desafío de tratar de minimizar los valores de *slowdown* particulares de cada proceso. El *throughput* será medido en la plataforma experimental a través de la métrica *STP* en función del *Average Normalized Turnaround Time.*

$STP = \sum_{i=0}^{N} ANTT_i$

Siendo N en el número de aplicaciones en ejecución simultáneamente en el sistema.

2.2 Definición de $BW_{slowdown}$

Calcular la degradación mediante la estimación de la métrica IPC es muy complejo. En su lugar, se toma la siguiente ecuación propuesta [5] para medir esta degradación en función del ancho de banda que consume un proceso y del que consumiría en caso de no existir competencia por los recursos compartidos del sistema.

$BW_{slowdown} = \frac{BW_{alone}}{BW_{shared}}$

Donde BW_{alone} y BW_{shared} son el de ancho de banda consumido por una aplicación, cuando no hay en ejecución ninguna otra tarea además de ésta en el sistema y cuando se ejecutan varios procesos simultáneamente con ella en el mismo sistema, respectivamente. Estos términos también serán citados con frecuencia en el presente documento.

Mientras que el valor de BW_{shared} puede ser obtenido mediante el soporte que proporciona PMCTrack [13, 16] de la tecnología Intel MBM [17], incluida en el procesador del sistema [25] sobre el que se trabaja, el cálculo de BW_{alone} es un reto significativo.

2.2.1 Definición de BW_{alone}

Cuando una aplicación se ejecuta simultáneamente con otras no es posible medir cuál sería el ancho de banda que consumiría en caso de ejecutarse sin éstas. Por ello, es preciso realizar una estimación.

Además de ofrecer una solución para estimar la degradación que sufren las aplicaciones se propone además un modelo [5] para obtener el valor de BW_{alone} en función del ancho de banda que consume una aplicación en un determinado periodo de tiempo y del ancho de banda consumido, por todas las aplicaciones que se encuentren en ejecución (BW_{total}), en ese mismo periodo de tiempo, además de la capacidad máxima del ancho de banda que el sistema puede abastecer (BW_{max}).

$BW_{total} = \sum_{i=0}^{N} BW_i^{shared}$

Para estimar el valor de BW_{alone} de un proceso que se ejecuta simultáneamente con otros en el mismo sistema se utiliza el siguiente modelo:

$b = \frac{u*(u+1-U)}{u*(u+1-U)+1-U}$

Donde:

$u = \frac{BW_{shared}}{BW_{max}}$

$U = \frac{BW_{total}}{BW_{max}}$

$BW_{alone} = b * BW_{max}$

2.2.1.1 Estudio de alternativas para el cálculo de BW_{alone}

Se evaluó también otra propuesta para realizar el cálculo de BW_{alone} utilizada en escenarios donde no es posible medir el consumo de ancho de banda basada en el número de fallos producidos en el último nivel de cache en las vías asociadas a la CPU donde se ejecuta la aplicación sobre la que se procede a realizar la estimación.

$BW_{alone} = \frac{LLC_MISSES * CACHE_LINE_SIZE}{ELAPSED_TIME}$

Para la evaluación, fue necesario realizar una selección de benchmarks del SPEC CPU 2006 que hacían uso intensivo de la memoria principal y se ejecutaron, uno tras otro, para obtener el valor real de ancho de banda consumido obtenido mediante el soporte de PMCTrack [13, 16] de la tecnología Intel MBM [17]. Después estas mismas aplicaciones de referencia (gemsfdtd, lbm, libquantum, mcf) se ejecutaron junto réplicas con el fin de poder validar la estimación.

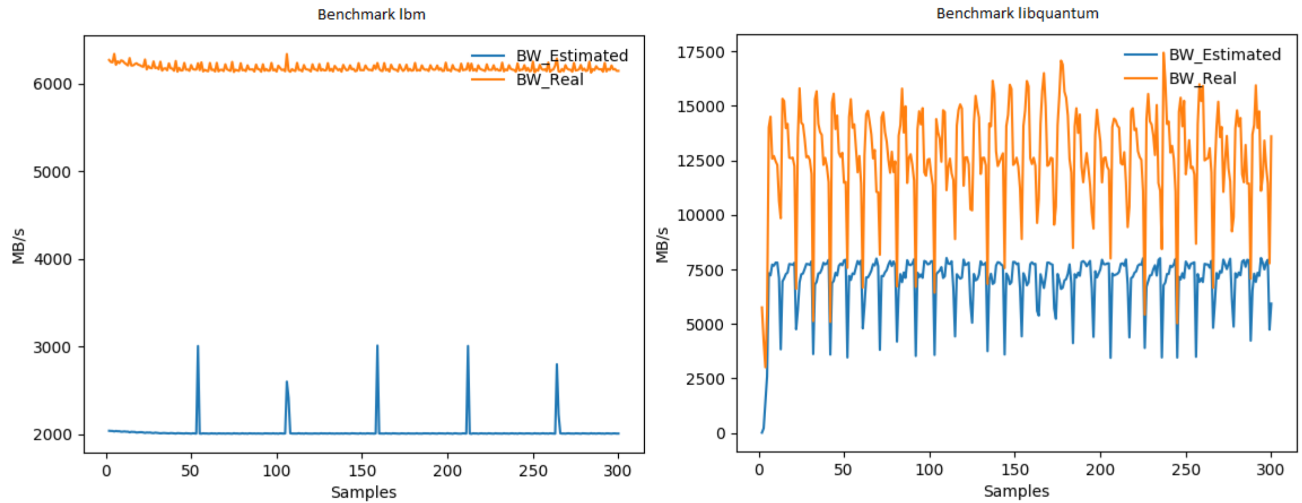


Figura 2.1: Cálculo alternativo de BW_{alone}

Cuando enfrentamos en gráficas ambos resultados [**fig. 2.1**], los datos medidos con el resultado de la ecuación propuesta en esta subsección, se puede observar que la estimación realizada se aleja del resultado obtenido de la medición real. Esta apreciación ha sido recurrente con muchos de los benchmarks con los que se ha realizado esta evaluación.

Este error cometido podría producirse en parte por no considerar el ancho de banda consumido cuando la memoria cache realiza la tareas de prefetching o en el momento de la operación write-back.

2.2.2 Evaluación de la estimación de slowdown mediante el cálculo de $BW_{slowdown}$

Para realizar esta evaluación se tomaron una serie de benchmarks del SPEC CPU 2006 (astar, lbm, libquantum, mcf, milc, soplex, xalancbmk, bwaves, gemsfdtd), de ahora en adelante serán benchmarks de referencia, favoreciendo la diversidad en el uso que hacen de los recursos del sistema.

Por cada benchmark se realizan N ejecuciones, siendo N el número de núcleos del procesador [25] que incluye la plataforma experimental:

- (1) Una primera ejecución donde se ejecuta la aplicación de referencia sola en el sistema.
- (2...N) N-1 ejecuciones introduciendo en el sistema cada vez una aplicación adicional a la ejecución anterior además de la de referencia.

El benchmark escogido para ejecutarlo conjuntamente con la aplicación de referencia es un benchmark sintético que introduce una carga de trabajo significativa en varios recursos compartidos del sistema [14, 26].

Gracias a la herramienta de línea de comandos de PMCTrack [13, 16] y un script escrito en Bash, pudieron realizarse todas las ejecuciones de manera sistemática para llevar a cabo este experimento. Esta herramienta permite al usuario visualizar los valores de las métricas necesarias para poder llevar a cabo la evaluación de las estimaciones citadas en este capítulo. Estos datos serán recogidos para su posterior análisis.

Conociendo el ancho de banda que consumen cada una de las aplicaciones de referencia cuando se ejecutan sin competir por los recursos, se pudo comparar este valor con el resultado de la ecuación propuesta para calcular BW_{alone}. Conociendo además el valor de IPC_{alone} así como el de IPC_{shared} de éstas, se evaluará si el cálculo de $BW_{slowdown}$ es equivalente al valor de *slowdown*.

Los resultados de la evaluación de ambas ecuaciones arrojaron conclusiones relevantes. En el eje de ordenadas nos encontramos los valores de *slowdown* para las distintas ejecuciones de una misma aplicación, de izquierda a derecha en el eje X, desde que la aplicación de referencia se ejecuta sin competencia hasta que se ejecutan simultáneamente N-1 aplicaciones intrusas además de ésta.

$BW_{slowdown}$ representa al valor de *slowdown* estimado por el modelo con el dato de ancho de banda consumido, cuando no hay competencia por los recursos, conocido y $BW_{slowdown__}BW_{alone_estimated}$ representa la estimación de *slowdown* habiendo calculado BW_{alone}. $IPC_{slowdown}$ representa el valor real de la degradación producida, *slowdown*.

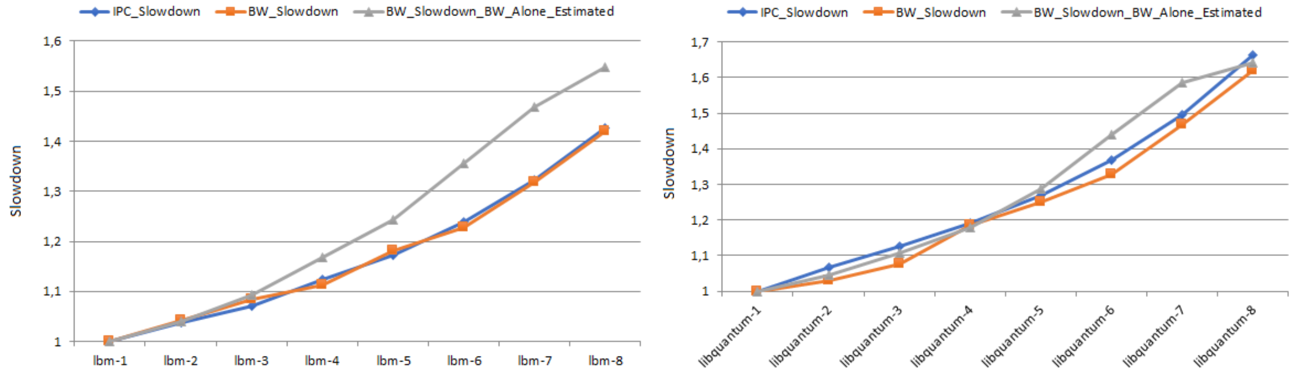


Figura 2.2: Evaluación positiva $BW_{slowdown}$

En la figura [**fig. 2.2**] se observa que la estimación de *slowdown* mediante el cálculo de $BW_{slowdown}$ es efectivo.

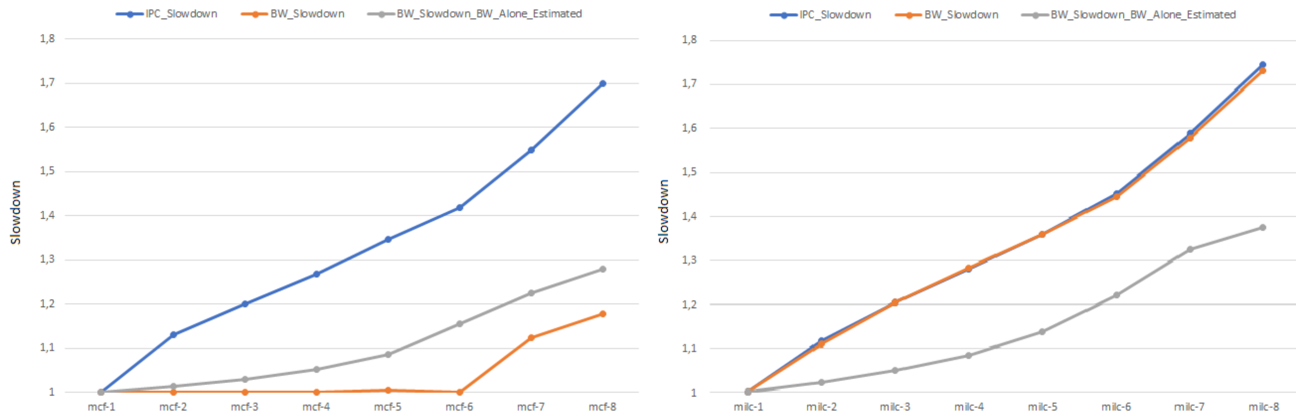


Figura 2.3: Evaluación negativa $BW_{slowdown}$

En este caso [**fig. 2.3**] pueden verse dos fenómenos distintos. En la imagen de la izquierda la predicción no es buena mientras que en la de la derecha sí es precisa. No obstante, el modelo de estimación de BW_{alone} no parece resultar válido en este segundo caso ya que la diferencia entre conocer el valor real del ancho de banda que consumiría una aplicación si no se ejecutase simultáneamente con ninguna otra y estimarlo es relevante a la hora de realizar el cálculo de $BW_{slowdown}$.

Los datos de *slowdown* resultantes de emplear las ecuaciones propuestas en [5] tienen cierta similitud con los resultados reales cuando las aplicaciones de referencia hacen un uso intensivo de la memoria principal.

No obstante, debido a que las aplicaciones pueden tener fases muy irregulares [**fig. 2.4**], en cuanto al uso de ancho de banda durante su ejecución, hay un conjunto de benchmarks de referencia que no proporcionaron resultados óptimos ya que se expuso a estudio la media, tanto del ancho de banda utilizado en el momento del experimento BW_{shared} para el cálculo de BW_{alone} como del valor real cuando el proceso en cuestión se ejecutaba sin competencia por los recursos.

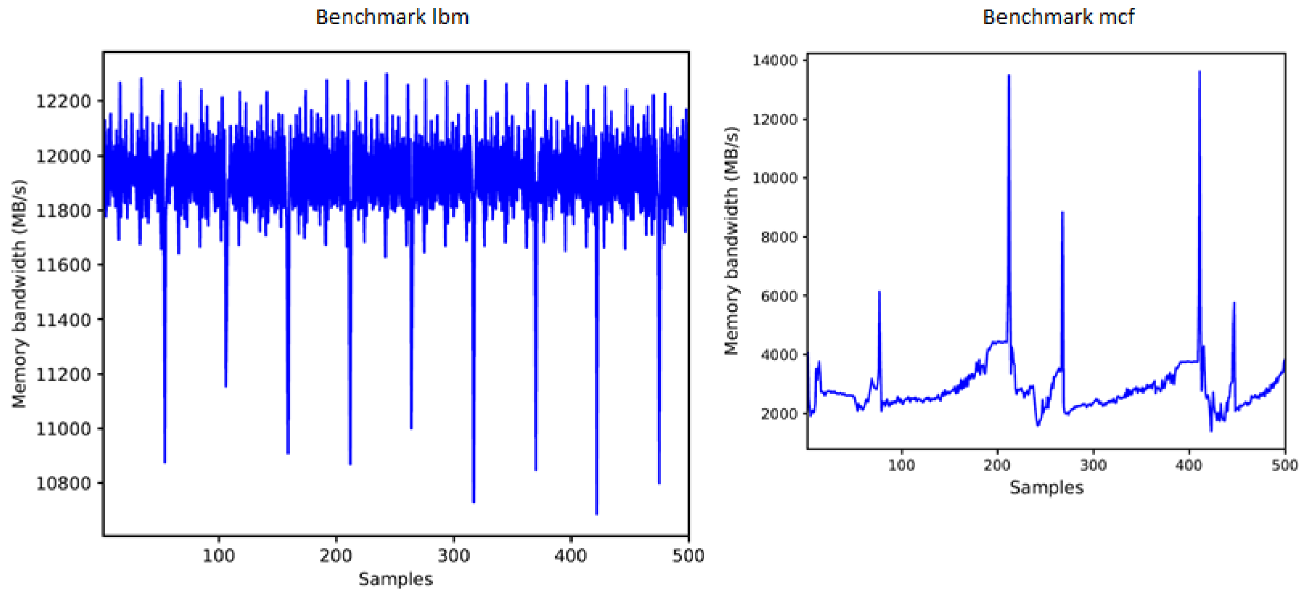


Figura 2.4: Ancho de banda requerido a lo largo del tiempo

Al considerarse la media de todas las variables que intervienen en la estimación de BW_{alone}, requisito para calcular $BW_{slowdown}$, de la aplicación de referencia existe la posibilidad de que la evaluación de la técnica propuesta no pueda considerarse como válida cuando se presentan estas fases irregulares de demanda de ancho de banda.

No obstante, en el momento que se presentan fases regulares el modelo de estimación de *slowdown*, incluido el del cálculo BW_{alone}, parece dar buenos resultados como para llevar a cabo su implementación.

2.3 Definición de unfairness

Ya se ha hecho mención acerca de cómo poder estimar el ancho de banda que las aplicaciones utilizarían en caso no tener que competir por los accesos a memoria principal cuando sí lo hacen BW_{alone}. Asimismo, se han presentado modelos [5] para estimar la degradación producida, debido a la competencia.

Es momento de recordar que el objetivo de este trabajo es priorizar la justicia por el uso de los recursos compartidos entre todas las aplicaciones en ejecución. La métrica *unfairness* se encuentra presente en artículos que forman parte de la bibliografía de este trabajo [10, 14] y se define tal que:

$unfairness = \frac{max(slowdown_0, slowdown_1, ..., slowdown_{N-1})}{min(slowdown_0, slowdown_1, ..., slowdown_{N-1})}$

La métrica *unfairness* es *lower-is-better*. Se considera en este contexto que un sistema es justo si la degradación que sufren todos los procesos que se encuentren en ejecución es similar entre ellos.

A continuación, se detallan las características de la plataforma experimental así como el soporte hardware del que se hace uso para tratar de reducir el valor de la métrica *unfairness*.

Capítulo 3

Plataforma experimental y extensiones del hardware

Para la realización de este trabajo se ha hecho uso de diversos recursos hardware y software. Se expone en la **sección no. 3.1** las características técnicas de la plataforma experimental utilizada así como las extensiones del hardware del procesador Intel [25] integrado en ésta en la **sección no. 3.2**. Por último, se presenta en la **sección no. 3.3** la herramienta de monitorización open source PMCTrack [13, 16] tratando de ilustrar las posibilidades que ofrece tanto al espacio de usuario como el soporte que proporciona en la implementación de módulos del kernel Linux para monitorizar métricas de rendimiento.

3.1 Características técnicas de la plataforma

Se ofrece un resumen en esta sección de las características técnicas de relevancia de la plataforma experimental [**tbl. 3.1**] que fue utilizada para la realización de este trabajo.

Tabla 3.1: Especificaciones de la plataforma experimental

Característica	Descripción
Modelo	Intel(R) Xeon(R) CPU E5-2620 v4
Número de cores	8
Frecuencia	2.1GHz
Memoria	DRAM 4 x 8GB DDR4 @ 2133 MHz
Máximo ancho de banda de memoria	68.3 GB/s

El procesador Intel(R) Xeon(R) CPU E5-2620 v4 [25] dispone de la siguiente jerarquía de memoria cache [**tbl. 3.2**]:

Tabla 3.2: Jerarquía de memoria cache del procesador Intel(R) Xeon(R) CPU E5-2620 v4

Nivel	Tamaño	Función de correspondencia	Política de escritura	Ámbito
L1I	8 x 32 KB	Asociativa por conjuntos - 8 vías	Write-back	Privada por core
L1D	8 x 32 KB	Asociativa por conjuntos - 8 vías	Write-back	Privada por core
L2	8 x 256 KB	Asociativa por conjuntos - 8 vías	Write-back	Privada por core
L3	20 MB	Asociativa por conjuntos - 20 vías	Write-back	Compartida

No se debe olvidar que la plataforma experimental ejecuta un Debian GNU/Linux 8 Jessie con una versión custom del kernel Linux 4.1.5 compilada para permitir el soporte que proporciona PMCTrack [13, 16].

En la siguiente sección se detallan las extensiones hardware que resultan relevantes de este procesador para el desarrollo de este trabajo.

3.2 Extensiones del hardware

La plataforma experimental cuenta con el procesador Intel(R) Xeon(R) CPU E5-2620 v4 [25]. Éste dispone de dos tecnologías [15, 17] de las que se ha hecho uso en particular.

Se presentan a continuación estas extensiones hardware así como el uso que se ha hecho de las mismas en este trabajo.

3.2.1 Intel CAT - Cache Allocation Technology

La tecnología Intel CAT [15] proporciona control software sobre la gestión del último nivel de cache.

El hardware por sí mismo no otorga a las aplicaciones una fracción de los recursos compartidos proporcional a la prioridad que establece el usuario. Esto supone un serio problema para el sistema operativo, ya que la contención por recursos compartidos puede afectar muy negativamente a la calidad del servicio que el sistema ofrece los usuarios de la plataforma. Intel proporciona soporte para el kernel Linux de la tecnología Intel CAT [27] permitiendo así poder establecer el uso que hacen las aplicaciones del último nivel de cache.

Debe explicarse el concepto *class of service* o *CLoS* [**fig. 3.1**]. Cada *CLoS* actúa como un registro que contiene información relativa a las vías de último nivel de cache que pueden ser utilizadas por el proceso asociada a ella. Es decir, mediante software es posible realizar asociaciones de procesos a determinadas vías con cualquier propósito.

El hardware disponible cuenta con 16 *CLoS*, siendo una de ellas reservada para el sistema operativo, con 20 bits cada una coincidiendo con las vías de cache de último nivel disponibles. En los casos en los que los bits de estos registros no se superponen, las aplicaciones no compiten entre sí por este recurso.

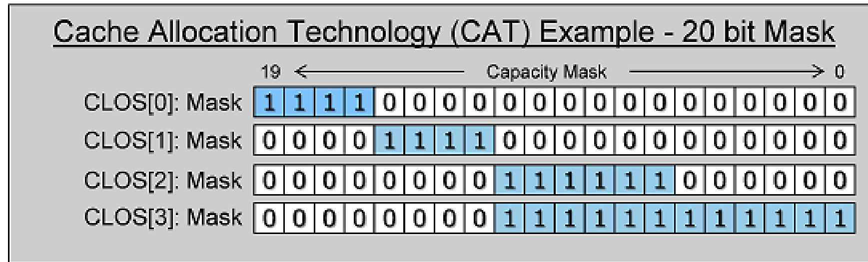


Figura 3.1: Intel CAT - Cache Allocation Technology

PMCTrack [13, 16] proporciona una interfaz que facilita el uso del soporte de Intel CAT [27] en el kernel Linux. Además, previamente a la realización de este trabajo, fue implementado un algoritmo que toma el control sobre las aplicaciones en ejecución para tratar de proporcionar a éstas de manera equitativa disponibilidad exclusiva de vías de último nivel de cache.

Para relacionar la implementación del algoritmo citado con la descripción de *CLoS*, cada aplicación en ejecución será asignada a una de estas. Tomando como premisa que se hará un uso total sobre el recurso del último nivel de cache, se proporcionará un número de vías igualitario. En caso de no ser posible ese reparto, las vías sobrantes serán proporcionadas a los distintos procesos durante un periodo de tiempo determinado favoreciendo, nuevamente, la justicia por el uso de los recursos compartidos.

3.2.2 Intel MBM - Memory Bandwidth Monitoring

La tecnología Intel MBM [17] se encuentra también presente en el procesador que incluye el sistema del que se hace uso en este trabajo.

La característica principal de la que se ha sacado partido ha sido la que permitía la posibilidad de monitorizar el ancho de banda en memoria requerido por cada proceso en ejecución. Esta métrica se tiene en cuenta en este proyecto para gestionar la utilización del ancho de banda disponible priorizando, nuevamente, la justicia en su utilización.

De partida, nuevamente gracias a una interfaz proporcionada por PMCTrack [13, 16], pudo contarse con el soporte que proporciona Intel en el kernel Linux de esta tecnología con un alto nivel de abstracción.

3.3 Herramienta de monitorización PMCTrack

La aportación de la herramienta open source de monitorización PMCTrack [13, 16] ha sido de gran relevancia para la realización de este trabajo. Deben ser distinguidos dos casos de uso de los que se ha hecho empleo. El primero de ellos, sobre el que versa esta sección, está relacionado con la fase relativa a la evaluación de los modelos [5] propuestos para permitir alcanzar el objetivo establecido en este trabajo, minimizar la métrica *unfairness*.

Esta herramienta muestra al usuario, a través de la interfaz de línea de comandos, métricas obtenidas a través de los contadores hardware existentes en la plataforma experimental, así como el ancho de banda consumido por un proceso en ejecución gracias al soporte que tiene de la tecnología Intel MBM [17]. Se ilustra a continuación la configuración previa y el caso de uso propuesto.

Mediante el uso de las siguientes instrucciones la herramienta se carga en el sistema operativo:

```
$ pmctrack-manager load-module
$ sudo modprobe msr
$ turbo-boost disable
```

Antes de la realización de este trabajo había ya desarrollados módulos del kernel Linux preparados para monitorizar distintos eventos hardware haciendo uso del soporte que proporciona esta herramienta. Ya cargada podemos tener a nuestra disposición cualquiera de estos módulos.

```
$ echo 'activate 2' > /proc/pmc/mm_manager
```

El módulo número 2, *PMCtrack module that supports Intel CMT*, tiene soporte de las tecnologías Intel CAT [15] e Intel MBM [17] y fue utilizado antes de la implementación del propuesto para llevar a cabo los experimentos asociados a validar los modelos considerados en este trabajo [5].

Se ilustra ahora el uso de esta herramienta, así como el formato en el que proporciona la información:

```
$ benchlocal@debussy:~/het-harness$ pmctrack -T 0.5 -b 3 -c instr,cycles -n 100
    /-V virt0 set_cos 1 ./benchmarks/common/leslie3d06
    /| pmc-metric -m bw='virt0/etime_us'
nsample    pid      event         pmc0         pmc1   etime_us                  bw
      1   4531       tick   2821991541    957807267     498497      1074934.317133
      2   4531       tick   3096624580   1046645357     499994         3388.514006
      3   4531       tick   3108016751   1046617291     500003         3409.817621
      4   4531       tick   3096048209   1046664034     500002         3382.496038
      5   4531       tick   3093800353   1046669767     499997         3406.057428
      6   4531       tick   3098721861   1046639554     499999         3374.324317
```

Donde:

- -b – Número de CPU que ejecutará el proceso.
- -c – Contadores hardware que se desea visualizar.
- -T – Time-Based Sampling. Tiempo en segundos transcurrido entre la toma de dos muestras.
- -V – Contadores virtuales.
- -n – Número de muestras a tomar.
- set_cos – El proceso en ejecución es asignado a una *CLoS*.

Pueden consultarse los distintos contadores disponibles mediante la siguiente instrucción:

```
$ pmc-events -L

[PMU 0]
instr_retired_fixed
unhalted_core_cycles_fixed
unhalted_ref_cycles_fixed
instr
cycles
unhalted_core_cycles
instr_retired
unhalted_ref_cycles
llc_references
llc_references.prefetch
llc_misses
llc_misses.prefetch
branch_instr_retired
branch_misses_retired
l2_references
l2_misses
l2_lines_in
[Virtual counters]
total_llc_bw
```

Estos contadores se encuentran configurados en la implementación del módulo del kernel Linux. Caso particular es el del contador virtual *total_llc_bw*; esta métrica no procede de un contador hardware si no que es el dato de ancho de banda consumido proporcionado por la tecnología Intel MBM [17] expuesto como un contador virtual.

Para obtener los códigos necesarios para realizar esta configuración basta con ejecutar el siguiente comando:

```
$ pmc-events instr,cycles,llc_misses
pmc0,pmc1,pmc3=0x2e,umask3=0x41
```

Y proporcionar a la configuración de PMCTrack [13, 16] este vector:

```
static const char* bw_sched_pmcstr_cfg[]= {
    pmc0,pmc1,pmc3=0x2e,umask3=0x41,NULL
};
```

En el siguiente capítulo se expone el segundo caso de uso de PMCTrack [13, 16] para la realización de este trabajo. Se presenta el diseño y la implementación del algoritmo de gestión de los recursos compartidos en tiempo real, así como la aportación de esta herramienta para hacerlo posible.

Capítulo 4

Diseño e implementación

En este capítulo se presentan con detalle los pasos previos que se llevaron a cabo antes de realizar el desarrollo en el kernel Linux, **sección no. 4.1**, así como los aspectos de mayor relevancia de éste una vez finalizado, **sección no. 4.2**.

En las siguientes secciones se hace distinción entre el desarrollo del módulo del kernel Linux, en el que que se usa el soporte de PMCTrack [13, 16] para la toma periódica de métricas, y la implementación de los algoritmos encargados de la gestión de los recursos compartidos en tiempo real propuesto en este trabajo.

Este algoritmo toma el papel de realizar con precisión las estimaciones presentadas para reducir la métrica *unfairness*.

4.1 Diseño del módulo del kernel Linux

Se presentan en esta sección los recursos utilizados por la implementación del módulo del kernel Linux con el fin de facilitar la comprensión del mismo en la sección donde se detalla su desarrollo. Además, se describen los algoritmos diseñados para afrontar el desafío de minimizar el *slowdown* de los procesos en ejecución con el fin de reducir el valor de la métrica *unfairness*.

4.1.1 Información almacenada por aplicación

Una aplicación está representada por la estructura *app*. Ésta contiene los siguientes campos:

- BW_{alone} representa la estimación de ancho de banda que consumiría la aplicación en caso de no tener que ejecutarse simultáneamente con otras.
- BW_{shared} es el ancho de banda consumido por la aplicación.
- BW_{cap} representa la cuota de ancho de banda asignada a consumir en el siguiente periodo.
- *weight* representa el peso que tiene una aplicación en el momento que se hace el reparto de ancho de banda a utilizar en el siguiente periodo.

- *step_counter* es el contador de periodos transcurridos desde que la aplicación inició su ejecución.
- $LIFE$ es el contador de progreso de la aplicación.
- $MaxLIFE$ es el progreso máximo que puede alcanzar una aplicación hasta el periodo en curso.
- *Slowdown* representa a $BW_{slowdown}$. Estimación de *slowdown* de la aplicación durante un periodo.
- *SlowdownAggregate* representa al valor de $BW_{slowdown}$ acumulado desde el inicio de la ejecución de la aplicación.

Todos los campos de la estructura *app* son actualizados al final de cada periodo salvo BW_{shared} que es el campo donde se acumulará el ancho de banda medido durante un determinado periodo cada vez que se toma una nueva muestra.

Algunos de ellos son la representación de las ecuaciones presentadas con anterioridad [no. 2]. No obstante, existen otros de los que es necesario especificar nuevos cálculos para su llevar a cabo su actualización al final de cada periodo.

El contador de progreso de la aplicación se calcula en base a la degradación que sufre a lo largo de su ejecución.

$SlowdownAggregate_{app} = \frac{MaxLIFE_{app}}{LIFE_{app}}$

$MaxLIFE_{app} = 100 * step_counter_{app}$

$LIFE_{app} = \sum_{i=0}^{step_counter_{app}} \frac{100}{Slowdown_i^{app}}$

La asignación de un peso a cada uno de los procesos activos se hace en función del uso que harían del ancho de banda en caso de no tener competencia por éstos y la degradación producida por este hecho. Se define además *total_weight* como la suma de los pesos de las N aplicaciones que se encuentran en ejecución a las que se asigna proporcionalmente una cuota de ancho de banda para usar en el siguiente periodo.

$total_weight = \sum_{i=0}^{N} weight_i$

$weight = BW_{alone} * BW_{slowdown}$

4.1.2 Diseño de algoritmos para reducir la métrica unfairness

Relevantes son las siguientes funciones:

determine_bw: Se utiliza para obtener el total de ancho de banda del que deben de hacer uso las aplicaciones que se encuentran en ejecución para que la de referencia alcance un *slowdown* objetivo, $slowdown_target$. [**Algoritmo 2**]

Del modelo donde se obtiene el valor de $BW_{slowdown}$ de un proceso [5] puede ser inferido el valor de BW_{shared}, una vez estimada la métrica BW_{alone}, que tendría que obtener la aplicación de referencia para poder alcanzarlo.

$BW_{shared}^{ref} = \frac{BW_{alone}^{ref}}{slowdown_target}$

Dado el requisito que tiene la aplicación de referencia puede obtenerse el consumo que deben hacer todos los procesos en ejecución del ancho de banda disponible, BW_{total}, para que ésta pueda alcanzar su objetivo de *slowdown*. Esta ecuación es obtenida mediante el modelo que permite realizar la estimación sobre la métrica BW_{alone} [5].

$b = \frac{BW_{alone}^{ref}}{BW_{max}}$

$u = \frac{BW_{shared}^{ref}}{BW_{max}}$

$U = 1 - \frac{(u)^2 * (\frac{1}{b} - 1)}{u * (1 - \frac{1}{b}) + 1}$

$BW_{total} = U * BW_{max}$

$BW_{remain} = BW_{total} - BW_{shared}^{ref}$

$determine_bw(BW_{max}, app, slowdown_target) \rightarrow (BW_{remain}, BW_{shared}^{app})$

determine_bw_cap: Realiza el reparto del ancho de banda restante, BW_{remain}, después de asignar a la aplicación de referencia el suficiente para que alcance el *slowdown* objetivo. [**Algoritmo 3**]

$determine_bw_cap(\vec{app}, BW_{remain}, total_weight) \rightarrow (\vec{BW}_{cap}^{app})$

Para llevar a cabo una asignación eficaz de cuotas de ancho de banda entre los procesos que se encuentran en ejecución se toman ideas de un estudio dedicado a distribuir de manera proporcional el uso de los recursos disponibles [9], en este caso proporcional en base al peso que disponga cada aplicación. El uso de esta técnica asegura que un proceso no tendrá más cuota de la que usaría y tratará de favorecer a aquellas aplicaciones que, o bien sufren mayor degradación o requieren de mayor cuota de ancho de banda. Se parte de la premisa de que este reparto hará posible que las aplicaciones que sufren en mayor medida la contención en los accesos a memoria principal podrán ser aceleradas en detrimento de aquellas, que también su progreso está basado en la disponibilidad de ancho de banda, que sufren de menor degradación al tener que competir por los recursos.

4.1.3 Diseño del comportamiento del módulo del kernel Linux

La descripción del comportamiento, basado en trabajos propuestos por otros autores [7, 10], se encuentra en el [**Algoritmo 1**]. Cada iteración sobre el bucle principal representa el cómputo que se lleva a cabo durante un determinado periodo en el sistema real. Dentro de este intervalo se producen lecturas, también de manera periódica aunque con un periodo inferior, sobre los contadores hardware para determinar si las aplicaciones han superado la cuota que les ha sido asignada. Superada esta cuota se procederá a detener la ejecución de la aplicación que la haya alcanzado.

Cuando el periodo en curso finalice, se procede a aplicar los algoritmos presentados en la subsección anterior. La finalidad es la de determinar la cuota de ancho de banda disponible que tendrán los procesos activos en el siguiente periodo, para tratar de reducir la métrica *unfairness*, además de reanudar la ejecución de las tareas sobre las que se hubiera impuesto una penalización.

Antes de la implementación final en el kernel Linux se llevó a cabo el desarrollo de un simulador en un lenguaje de alto nivel, Python. Teniendo disponibles los datos de ancho de banda utilizado por cada uno de los benchmarks del SPEC CPU 2000 y del SPEC CPU 2006 pudo simularse el comportamiento que tendría la aplicación de los algoritmos propuestos en esta sección.

La principal finalidad de este software es la de facilitar la asimilación de todos los conceptos presentados para que su implementación en el kernel Linux sea más sencilla. La ventaja de utilizar un lenguaje de alto nivel como Python es la posibilidad que ofrece éste de usar herramientas de depuración para detener la ejecución del simulador en cualquier momento para poder tomar decisiones de implementación. Destacar la posibilidad de usar estos mecanismos es relevante ya que en el kernel Linux los fallos de programación pueden transformarse en cuelgues del sistema operativo; lo que supone un gasto en tiempo muy elevado en este punto donde se está terminando por definir el comportamiento que tendrá el módulo.

Una vez finalizado el diseño del comportamiento de éste, así como el de los algoritmos encargados de reducir la métrica *unfairness* se procede a llevar todos estos conceptos a la implementación en el kernel Linux.

Algoritmo 1: Comportamiento del módulo del kernel Linux.

```
1  function Comportamiento
   Input: threshold
2  begin
3      begin
4          Inicialización.
5          N ← |app⃗|
6          for app ∈ app⃗ do
7              BW_cap^app ← ∞
8              step_counter_app ← 0
9              LIFE_app ← 0
10         end
11     end
12     begin
13         Cómputo.
14         while N do
15             Mientras que no se cumpla el periodo...
16             for app ∈ app⃗ do
17                 Acumular el ancho banda consumido en BW_shared^app durante este periodo.
18                 if BW_shared^app == BW_cap^app then
19                     ¡Detener la tarea!
20                 end
21             end
22             for app ∈ app⃗ do
23                 ¡Reanudar la ejecución de la tarea si ha sido detenida!
24                 step_counter_app+=1
25                 Actualizar SlowdownAggregate_app.
26             end
27             Ordenar app⃗ por LIFE_app ascendente.
28             if N > 1 then
29                 if (LIFE_{N-1}^app − LIFE_0^app) > threshold then
30                     total_weight ← Σ_{j←2}^N SlowdownAggregate_j * BW_alone^j
31                     slowdown_target ← estimate_slowdown_target(app⃗)
32                     BW_remain ← determine_bw(app_0, slowdown_target)
33                     app⃗_tbc ← app⃗ \ app_0
34                     Ordenar app⃗_tbc por weight_tbc ascendente.
35                     determine_bw_cap(app⃗_tbc, BW_remain, total_weight)
36                 end
37             end
38         end
39     end
40 end
```

Algoritmo 2: Algoritmo encargado de establecer el uso que debe hacerse del ancho de banda para que una aplicación alcance un slowdown objetivo.

```
1  function determine_bw
   Input  : app, slowdown_target
   Output: BW_remain
2  begin
3  |   BW_required ← BW_alone^app / slowdown_target
4  |   BW_total ← 1 − ((BW_required)^2 * (1/BW_alone^app − 1)) / (BW_required * (1 − 1/BW_alone^app) + 1)
5  |   BW_cap^app ← ∞
6  |   BW_remain ← BW_total − BW_required
7  |   return BW_remain
8  end
```

1 **function** determine_bw
 Input : $app, slowdown_target$
 Output: BW_{remain}
2 **begin**
3 $BW_{required} \leftarrow \frac{BW^{app}_{alone}}{slowdown_target}$
4 $BW_{total} \leftarrow 1 - \frac{(BW_{required})^2 * (\frac{1}{BW^{app}_{alone}} - 1)}{BW_{required} * (1 - \frac{1}{BW^{app}_{alone}}) + 1}$
5 $BW^{app}_{cap} \leftarrow \infty$
6 $BW_{remain} \leftarrow BW_{total} - BW_{required}$
7 **return** BW_{remain}
8 **end**

Algoritmo 3: Algoritmo encargado de hacer el reparto proporcional, en base al peso que disponga cada aplicación, del ancho de banda disponible.

1 **function** determine_bw_cap
 Input : $\vec{app}, BW_{remain}, total_weight$
2 **begin**
3 **for** $app \in \vec{app}$ **do**
4 $BW_{fair} \leftarrow BW_{remain} * \frac{SlowdownAggregate_{app} * BW^{app}_{alone}}{total_weight}$
5 $BW_{peak} \leftarrow BW^{app}_{alone}$
6 $BW^{app}_{cap} \leftarrow min(BW_{remain}, min(BW_{fair}, BW_{peak}))$
7 $total_weight$-=$SlowdownAggregate_{app} * BW^{app}_{alone}$
8 BW_{remain}-=BW^{app}_{cap}
9 **end**
10 **end**

4.2 Implementación del módulo del kernel Linux

En esta sección se describe con alto nivel de detalle el desarrollo del módulo del kernel Linux, así como la implementación de los algoritmos diseñados para reducir la métrica *unfairness*.

El desarrollo se ha realizado de la manera más flexible posible para poder implementar, con poco esfuerzo, cualquier técnica que pueda influir sobre la utilización de los recursos disponibles. Además, la implementación está dotada de un sistema de muestreo de la información almacenada de cada aplicación que se encuentra en ejecución. Esta utilidad permite validar las estimaciones abordadas en este trabajo.

En las siguientes subsecciones se presentarán los detalles técnicos de mayor relevancia de la implementación.

4.2.1 Implementación de la interfaz de PMCTrack para módulos del kernel Linux

Se describirá en primer lugar el comportamiento de cada una de las funciones que compone la interfaz de PMCTrack [13, 16]. Preciso es inicializar en primera instancia una estructura de datos *monitoring_module_t* indicando qué manejadores de eventos dispone el módulo desarrollado en este trabajo.

```
monitoring_module_t bw_sched_mm = {
    .info=MY_MODULE_STR,
    .enable_module=bw_sched_enable_module,
    .disable_module=bw_sched_disable_module,
    .on_read_config=bw_sched_on_read_config,
    .on_write_config=bw_sched_on_write_config,
    .on_fork=bw_sched_on_fork,
    .on_new_sample=bw_sched_on_new_sample,
    .on_exit=bw_sched_on_exit,

    [...]
};
```

Mientras que algunas funciones de la interfaz de PMCTrack [13, 16] permiten la posibilidad de realizar la configuración de los elementos que componen el módulo requeridos por la herramienta de monitorización para su correcto funcionamiento, al mismo tiempo que se habilita el soporte de la tecnología Intel CAT o se lleva a cabo la gestión del sistema de particionado de cache por software, otras tienen mayor relevancia en la implementación de los algoritmos asociados a la estimación de *slowdown* y a la reducción de la métrica *unfairness*.

Todas ellas serán descritas posteriormente en las siguientes subsecciones siguiendo la misma distinción.

4.2.1.1 Configuraciones del módulo del kernel Linux

Habilitar un punto de entrada/salida al módulo mediante una entrada en el pseudo-sistema de ficheros */proc* ha permitido que el comportamiento de éste pueda ser configurable sin necesidad de que tenga que ser deshabilitado para que pueda adoptarlo. La mayor ventaja que ha ofrecido se ha presentado en la etapa de experimentación donde el módulo tenía, en cada momento, diferentes configuraciones asociadas a su comportamiento.

Se han implementado para ello las funciones (callbacks) *bw_sched_on_read_config* para la salida y *bw_sched_on_write_config* para la entrada de datos. Estas interfaces se exponen en el fichero */proc/pmc/config*. Los parámetros configurables son los siguientes:

- **sched_period**: Periodo de planificación. Precisión en milisegundos.
- **enable_bw_scheduler**: Habilita o deshabilita la planificación del uso que se realiza sobre el ancho de banda disponible.
  - (0) Deshabilitado
  - (1) Habilitado
- **throttling_mode**: El uso de este parámetro es relativo a cómo proceder en el momento que una aplicación ha consumido su cuota de ancho de banda asignada.
  - (0) No se impone ninguna penalización.
  - (1) Detener a la tarea que haya superado su cuota de ancho de banda asignada.
  - (2) Detener a la tarea que haya superado su cuota de ancho de banda asignada salvo si es la única que se está ejecutando durante el periodo en curso.
  - (3) Cuando la mitad de las tareas son detenidas se inicia un nuevo periodo.

Se presentan a continuación los distintos casos de uso de la entrada */proc/pmc/config*:

- Para habilitar el uso de los algoritmos contenidos en el módulo:

  ```
  $ echo "enable_bw_scheduler 1" > /proc/pmc/config
  ```

- Si queremos que la planificación detenga a las tareas en ejecución que consuman su cuota de ancho de banda asignada, salvo la última:

  ```
  $ echo "throttling_mode 2" > /proc/pmc/config
  ```

- Para establecer que el periodo sobre el que se hará la planificación será de 10000 milisegundos:

  ```
  $ echo "sched_period 10000" > /proc/pmc/config
  ```

- Si se desea consultar el valor de los parámetros de configuración basta con escribir el siguiente comando:

  ```
  $ cat /proc/pmc/config

  sched_sampling_period=1000
  sched_period=10000
  enable_bw_scheduler=1
  throttling_mode=1
  ```

Adicionalmente, PMCTrack [13, 16] dispone también de parámetros de configuración a los que se tiene acceso a través de */proc/pmc/config* de manera análoga a los expuestos por el módulo desarrollado para este trabajo. El de mayor relevancia es la entrada que permite modificar el periodo con el que se realizará la toma de muestras de los datos asociados a los contadores hardware. Su identificador es *sched_sampling_period*; éste puede verse en la salida producida al ejecutar la operación de lectura del fichero de configuración del módulo cuyo valor es de 1000 milisegundos.

4.2.1.2 Captura y seguimiento de procesos en ejecución

En esta sección se describirá la implementación de las funciones *bw_sched_on_fork* y *bw_sched_on_new_sample*.

Ambas actúan a modo de callbacks:

bw_sched_on_fork: Esta función es invocada cuando se crea un nuevo proceso en el sistema. Cabe mencionar que esta implementación es la encargada de reservar memoria para la estructura de datos que almacena la información del proceso en ejecución. Esta estructura contendrá la información necesaria para realizar las estimaciones que contempla este desarrollo al mismo tiempo que se almacenan aquellas que intervendrán en la asignación de los recursos disponibles.

bw_sched_on_new_sample: Esta función se invoca en el momento que una nueva muestra de los contadores hardware ha sido recogida por PMCTrack [13, 16]. En este momento es cuando se almacenan datos relativos al proceso ejecutado por la CPU donde se ha tomado la muestra. Es esta función la encargada de comprobar si la tarea en ejecución ha superado la cuota de ancho de banda asignada haciendo uso de la tecnología Intel MBM [17] en el periodo en curso; si esto sucediera se activan mecanismos para detenerla.

Los dos callbacks son ejecutados en la CPU asignada al proceso donde se produce el evento en cuestión no interfiriendo en la ejecución del resto de tareas que se encuentran en ejecución.

4.2.2 Implementación de algoritmos en el módulo del kernel Linux

Mientras que la configuración del módulo, la captura y el seguimiento de procesos en ejecución es gestionado por las funciones implementadas de la interfaz de PMCTrack [13, 16] destinadas para cada fin, el comportamiento del módulo queda delegado en la implementación de las ecuaciones y los algoritmos ya descritos en la sección anterior.

Habiéndose hecho mención de los callbacks *bw_sched_on_read_config*, *bw_sched_on_write_config*, *bw_sched_on_fork* y *bw_sched_on_new_sample* queda por citar el uso de las funciones *bw_sched_enable_module* y *bw_sched_disable_module*, también de relevancia.

La función *bw_sched_enable_module* realiza las operaciones imprescindibles para que el módulo pueda ser cargado. Se establecen aquí los valores por defecto en los recursos del sistema operativo de los que se hace uso y se reserva memoria para las estructuras de datos

requeridas. En este momento se activa el soporte de la tecnología Intel CAT además de realizar la configuración requerida por PMCTrack [13, 16]. En contraposición, la implementación de la función *bw_sched_disable_module* está destinada a liberar toda la memoria requerida por todos los recursos que forman parte del módulo.

Del mismo modo que existe una entrada que permite acceder a la configuración del módulo activo, */proc/pmc/config*, existe otra, */proc/pmc/mm_manager*, que permite activar bajo demanda cualquier módulo configurado para obtener el soporte que proporciona PMCTrack [13, 16]. Para activar cualquiera de ellos basta con ejecutar los siguientes comandos:

```
$ 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 - BW Cap monitoring module
[ ] 4 - BW Scheduler monitoring module
[ ] 5 - Oracle CA estimation model
[ ] 6 - Sched prototype monitoring module (CAT)

$ echo activate 4 > /proc/pmc/mm_manager
```

Donde el número 4 corresponde al módulo desarrollado para este trabajo. El nombre que aparecerá en esta entrada es configurable; éste se encuentra definido en *MY_MODULE_STR*.

Una vez activo, queda establecido un periodo, superior al de toma de muestras, gestionado mediante el uso de un temporizador, *hr_timer*, del kernel Linux. Al inicio de cada periodo se establece un tiempo de vencimiento; cada vez que un periodo vence se ejecuta una función donde se llevan a cabo las siguientes operaciones, en el siguiente orden:

1. Se realiza una copia de las muestras obtenidas de los contadores hardware por aplicación. El vencimiento de un periodo no implica que las aplicaciones en ejecución se detengan por lo que se antoja imprescindible salvar estos datos antes de que sean sobrescritos.
2. Se establece un nuevo periodo de vencimiento. A partir de este punto las muestras obtenidas forman del nuevo periodo. Al mismo tiempo, se reanuda la ejecución de las aplicaciones que hayan sido detenidas con anterioridad.
3. Con la información obtenida en el periodo anterior, para cada aplicación en ejecución se lleva a cabo la estimación de la métrica BW_{alone} y, a su vez, se realiza el cálculo de $BW_{slowdown}$, es decir, la estimación del valor de *slowdown*. En este momento se actualiza el valor del *slowdown* acumulado en función de las estimaciones realizadas.
4. Se aplican los algoritmos, **sección no. 4.1.2**, que determinan cómo puede el proceso que sufre mayor degradación alcanzar el *slowdown objetivo* y cómo deberá repartirse el ancho de banda disponible entre el resto de tareas. El tiempo que puede estar en ejecución un proceso en el actual periodo antes de ser detenido va determinado por el ancho de banda que usó en el periodo anterior y por el que se le asigna para que use durante el mismo.

Puede consultarse el comportamiento general en la **sección no. 4.1.3**.

El evento producido por el vencimiento del temporizador es atendido siempre por la CPU maestra; la asignación de esta queda delegada en el sistema operativo en el momento de activar el módulo. Se ha observado que la ejecución de los procedimientos citados anteriormente siempre por una misma CPU, la CPU maestra, no causa impacto en el rendimiento del proceso que tenga asociado.

Debe mencionarse que la decisión de realizar estos pasos en el orden expuesto es producto de la experimentación. Ejecutar el paso 2 antes de realizar las estimaciones y llevar a cabo la toma de decisiones que afecten al uso de los recursos del sistema se debe a que el periodo de toma de muestras no varía, por tanto, de no establecer un nuevo periodo de vencimiento con la máxima premura se podrían producir desajustes en la temporización de los eventos que traerían consigo estimaciones menos precisas. Cabe destacar, además, que las operaciones 3 y 4 siempre finalizan antes de la siguiente toma de muestras por lo que, de ser necesario detener a una tarea después de tomar la primera muestra podría realizarse.

En el momento que se detecta que una aplicación ha consumido su uso asignado de ancho de banda para el periodo vigente se activa una interrupción que hace ejecutar un callback en la CPU donde ha ocurrido el evento. Este callback es el encargado de imponer la penalización descrita en la configuración del módulo al proceso asociado. El intervalo de tiempo transcurrido entre la detección y la detención no es significativo.

Dado que pueden producirse distintos eventos de manera simultánea, eventos que acceden a las mismas direcciones de memoria para hacer operaciones de lectura y escritura, se antoja imprescindible el uso de cerrojos con el fin de realizar conjuntos de operaciones sobre las estructuras de datos presentes de manera atómica. Además, estos conjuntos no pueden ser interrumpidos por lo que las interrupciones quedarán deshabilitadas, si no lo estaban ya, para ser tratadas posteriormente. Proporcionado por el kernel Linux se hace uso del recurso *spinlock* para establecer secciones críticas en el contexto de procesadores multicore con memoria compartida.

4.2.2.1 Información almacenada por aplicación. Estructura app_t

La interfaz de PMCTrack [13, 16] da soporte para almacenar datos asociados a un proceso que se encuentra en ejecución. Existe disponibilidad de acceso a ellos en el callback *bw_sched_on_new_sample*, así como a los valores de los contadores hardware, por lo que la mejor idea posible para guardar la información relativa a métricas es hacerlo durante la ejecución de este manejador. Todas ellas se almacenan en una estructura de datos *app_t*.

Además de los datos que se recogen durante la toma de muestras, la aplicación de los algoritmos implementados requiere del uso de campos adicionales relevantes en el momento que se lleva a cabo la asignación de los recursos disponibles. Algunos de ellos fueron presentados en la **sección no. 4.1.1** mientras que otros han sido añadidos una vez llevado el diseño al entorno real en el kernel Linux. Se expondrán estos nuevos campos dividiéndolos en función de su utilidad:

- Datos relativos a identificar el proceso:
  - **name**: Identificador del proceso obtenido del campo usado para este fin en la estructura *struct task_struct* de las fuentes del kernel Linux.
  - **switch_in_ktime**: Marca de tiempo en la que se produjo el último cambio de contexto.
  - **last_cpu**: CPU donde se ejecuta el proceso.
  - **nr_ways**: Vías de último nivel de cache asignadas al proceso.
- Datos relativos a la toma de muestras:
  - **samples_dyn**: Vector. Contiene información relativa a cada muestra que se toma durante un periodo. Se almacena aquí el valor de BW_{shared} además de los coeficientes de regresión α y β de los que se dará detalle más adelante en la **sección** [**no. 5.3.1**].
  - **samples**: Vector. La información de *samples_dyn* es volcada aquí al final de cada periodo para ser procesada.
  - **samples_limit**: Cuando se tome la enésima muestra, durante el periodo vigente, establecida en este campo se detendrá la ejecución del proceso.
  - **samples_period**: Contador de muestras tomadas en el periodo en curso.
- Datos relativos al estado de un proceso:
  - **cpu_intensive**: Flag que indica que la aplicación tiene un consumo de ancho de banda muy bajo. Estas tareas nunca serán detenidas.
  - **skip**: Flag que excluye a un proceso de cualquier planificación. Las primeras muestras tomadas de los contadores hardware asociados a la ejecución de un nuevo proceso no contienen datos válidos por lo que se debe tener en consideración durante la asignación de los recursos disponibles y en las estimaciones realizadas.
  - **throttled**: Indica que la aplicación ha sido detenida al haber alcanzado la cuota de ancho de banda asignada.

Los campos $SlowdownAggregate$ y $LIFE$ son actualizados a través de los datos obtenidos por cada muestra en la implementación final incrementando el contador *step_counter* por cada una de ellas. Por tanto, aquellos de los que dependen lo hacen de la misma forma.

Adicionalmente, se calcula y se almacena la media aritmética de las muestras recogidas de aquellos campos requeridos para llevar a cabo la asignación de ancho de banda a consumir durante el siguiente periodo, **sección no. 4.1.2**, para todos los procesos en ejecución.

4.2.3 Implementación de sistema de logs con ftrace

En el instante que el periodo en curso vence, después de procesarse los datos relativos a las muestras tomadas durante éste, se genera un resumen de los resultados de las métricas obtenidas y estimadas para cada aplicación. Concretamente, se muestra la información almacenada en la estructura *app_t* correspondiente a cada proceso. Este resumen puede proporcionarse al espacio de usuario gracias a la herramienta de depuración del kernel Linux *ftrace* [23].

Entre las funcionalidades de *ftrace* se incluye la posibilidad de encolar mensajes en un buffer mediante la llamada a la función *trace_prink*; buffer al que se tendrá acceso a través del fichero */sys/kernel/debug/tracing/trace*. Opcionalmente, se puede acceder al fichero */sys/kernel/debug/tracing/trace_pipe* para que el buffer vaya proporcionando los datos según disponga de ellos.

```
$ cat /sys/kernel/debug/tracing/trace_pipe | grep "vim"
```

********** LIST SAMPLING **********
********** T. MODE: NO THROTTLE LAST APP **********

APP:lbm_base.linux- CPU:0 WAYS:2 T:0 I:0 SAMPLES:10 BW_ALONE:14633446 BW_SHARED:9619616 BW_TOTAL:37331392
LIFE:740 SLOWDOWN:1.340 SLOWDOWN_AGGR:1.351 A:318 B:672 STEP:10 STEPS_THROTTLED:0

APP:equake_base.lin CPU:1 WAYS:2 T:0 I:0 SAMPLES:10 BW_ALONE:8266354 BW_SHARED:5776448 BW_TOTAL:37331392
LIFE:796 SLOWDOWN:1.247 SLOWDOWN_AGGR:1.256 A:571 B:473 STEP:10 STEPS_THROTTLED:0

APP:fma3d_base.linu CPU:3 WAYS:3 T:0 I:0 SAMPLES:10 BW_ALONE:3857747 BW_SHARED:3026080 BW_TOTAL:37331392
LIFE:893 SLOWDOWN:1.106 SLOWDOWN_AGGR:1.119 A:730 B:295 STEP:10 STEPS_THROTTLED:0

[...]

********** END OF LIST SAMPLING **********

[...]

Se ilustra en este resumen que la información suministrada es la almacenada en la estructura de datos app_t asociada a un proceso; aquella que resulta más relevante. Cabe destacar que se mostrarán siempre los resultados obtenidos de haber procesado la última muestra del periodo que acaba de finalizar con el fin de no sobrecargar al sistema.

Además de proporcionar esta información de manera periódica el módulo insertará en el buffer de *ftrace* información de interés para la fase experimental cuando una aplicación finaliza su ejecución, es decir, cuando el callback *bw_sched_on_exit* es invocado.

```
cat /sys/kernel/debug/tracing/trace_pipe | grep "EOE"
```

APP:gap_base.linux- EXIT_CODE:0 STEP:40 ACTIVE_TIME:49999960706 PERIOD:10000000000 STEP_THROTTLED:4 PC:3339 SLOWDOWN_AGGR:314038 BW_ALONE_TOTAL:96284538 BW_SHARED_TOTAL:78786656 NR_WAYS_ACUM:80

[...]

Una vez finalizada la fase de implementación, estos datos serán utilizados en la fase experimental para validar en primera instancia las estimaciones realizadas, los cálculos de $BW_{slowdown}$ y de BW_{alone}. Posteriormente, se evaluará si los algoritmos propuestos en este capítulo permiten reducir la métrica *unfairness*.

Capítulo 5

Evaluación experimental

Una vez finalizada la etapa de implementación se presenta la fase experimental de este trabajo. La finalidad de ésta es la de poder confirmar que pueden hacerse estimaciones precisas, acerca de la degradación que sufren los procesos en ejecución, que permitan tratar de reducir la métrica *unfairness* haciendo uso de las técnicas expuestas en los capítulos anteriores.

Esta fase experimental se divide en 4 etapas:

1. Configuración del framework Het-Harness. Entorno que permite activar el módulo del kernel Linux desarrollado, así como modificar su configuración, y la ejecución de tareas bajo demanda. **Sección no. 5.1**.

2. Evaluación de la implementación de los algoritmos destinados a estimar la degradación que sufren los procesos en ejecución. **Sección no. 5.2**.

3. Estudio de alternativas para precisar la estimación de la degradación. **Sección no. 5.3**.

4. Presentación de los resultados obtenidos después de aplicar los algoritmos diseñados para reducir la métrica *unfairness*. **Sección no. 5.4**.

Las distintas etapas serán descritas en las siguientes secciones exponiendo los puntos de mayor interés de cada una de ellas.

5.1 Framework Het-Harness

Para llevar a cabo el proceso de experimentación se hará uso del framework Het-Harness desarrollado por miembros del departamento de Arquitectura de Computadores y Automática de la Facultad de Informática de la Universidad Complutense de Madrid.

Het-Harness permite lanzar cargas de trabajo de manera sistemática gracias a un conjunto de herramientas formadas por scripts, código en C y ficheros de configuración. La principal utilidad del framework es la capacidad de reportar el resultado de la ejecución de una carga de trabajo.

Una carga de trabajo es un conjunto de aplicaciones que se ejecutan simultáneamente en el sistema. Cuando una tarea incluida en una carga de trabajo específica finaliza su ejecución se lanza automáticamente una nueva réplica de ésta de tal forma que el uso de los recursos se mantiene constante hasta que la que necesita más tiempo para completarse ejecuta todas sus instrucciones; momento en el que se detienen el resto de los procesos activos. Sólo será considerado el tiempo de ejecución de una tarea si ésta llega a su fin por sí sola.

El resultado de cada ejecución es ofrecido al usuario en forma de fichero de texto; éste da información sobre el tiempo de ejecución de cada uno de los benchmarks que componen una carga de trabajo. Esto permitirá evaluar, en primera instancia, la precisión del cálculo de $BW_{slowdown}$ respecto al valor real de *slowdown* y del ancho de banda consumido cuando una aplicación no tiene competencia por los recursos, BW_{alone}. Posteriormente, estos datos serán utilizados para verificar si se produce una reducción en el valor de la métrica *unfairness* de las cargas de trabajo lanzadas.

Se presenta a continuación la configuración, el uso y la aportación de Het-Harness a este trabajo.

5.1.1 Configuración de Het-Harness

Para poder llevar a cabo el lanzamiento de experimentos es imprescindible tener disponibles una serie de scripts escritos en Bash y otros recursos de los que se hará mención a continuación. En los siguientes puntos se dará información sobre el papel que desempeñan cada uno de estos elementos.

5.1.1.1 Script de lanzamiento de experimentos

El script que se presenta en esta sección es el de mayor relevancia. En él se establece la ubicación de todos los ficheros requeridos por el framework para su correcto funcionamiento.

```
#!/usr/bin/bash

. ${HET_HARNESS_ROOT}/test_scripts/intel-cmt/multiapp_tests.common.sh

function bw_sched_config() {
  pmon_writepar "sched_period $1"
  pmon_writepar "sched_sampling_period $2"
  pmon_writepar "throttling_mode $3"
  pmon_writepar "enable_bw_scheduler $4"
  export tag=${btag}period_$1_sampling_$2_tmode$3_bwsched$4
}

function bw_sched(){
  stock_linux
  activate_mm 4
  export PART_BINDING="CPU"
  export NR_PARTITIONS=1
  export PART0="0 1 2 3 4 5 6 7"
  export HARNESS_PMCTRACK=yes
  export HARNESS_CALLBACK="${HET_HARNESS_ROOT}/test_scripts/intel-cmt/lanzador_experimen
  tag=".workload_"
}

experiments=(bw_sched)

cfg_file=${HET_HARNESS_ROOT}/cfg/quickia/ctimes.default.niter.cfg
runfiles_dir="${HET_HARNESS_ROOT}/benchsets"

spec_files=(${runfiles_dir}/intel-cat/bw_sched_workloads.spec)
prio_files=('bw_sched_config 530 52 2 1' 'bw_sched_config 530 52 0 0')

function restore_defaults_custom()
{
  restore_defaults
  export HARNESS_SAMPLES=2
  export HARNESS_TIMEOUT=200
}

run_experiments $1
```

Han de ser proporcionados los siguientes recursos:

- Fichero de configuración del framework, *cfg_file*. La configuración utilizada en este trabajo es la que trae por defecto.
- Fichero o ficheros con las cargas de trabajo que van a ser ejecutadas, *spec_files*.

Por otro lado, se describen los aspectos de mayor relevancia para este trabajo de la implementación de este script:

- Módulo del kernel Linux a activar, soportado por PMCTrack [13, 16], antes de la ejecución de una carga de trabajo. Se activa mediante el comando *activate_mm* en la función *bw_sched*; siendo el número 4 el identificador numérico del módulo desarrollado en este trabajo.
  - Puede establecerse la ubicación de un script escrito en Bash donde se encontrarán implementados callbacks que serán ejecutados antes y después de la ejecución cada carga de trabajo, *HARNESS_CALLBACK*.
  - Para esta fase experimental cada aplicación ejecutada se asignará a una CPU en particular. Este comportamiento viene definido en las variables *PART_BINDING*, *NR_PARTITIONS* y *PART0*.
- Se declara una función, *bw_sched_config*, con el propósito de escribir en la entrada */proc* del módulo activo, mediante el comando *pmon_writepar*, para modificar los parámetros de configuración que éste dispone.
- La variable *tag*, presente en las funciones *bw_sched* y *bw_sched_config*, permite añadir información adicional al fichero de log resultante de la ejecución de una carga de trabajo.

Para comprender cómo Het-Harness interpreta este script se expone el siguiente pseudocódigo:

```
for scheduler in experiments: # Por cada módulo.
  for config in prio_files: # Por cada una de las configuraciones.
    for spec_file in spec_files: # Por cada uno de los ficheros de cargas de trabajo.
      for workload in spec_file: # Por cada carga de trabajo.
        on_init() # Callback pre-ejecución.
        exec() # Ejecución.
        on_end() # Callback post-ejecución.
```

5.1.1.2 Script con la implementación de callbacks

Het-Harness da la posibilidad de establecer callbacks que son invocados cuando se lanza una carga de trabajo y cuando ésta finaliza su ejecución.

El callback *on_start_test()* se invoca antes proceder a la ejecución de las aplicaciones que componen una carga de trabajo; se utiliza para eliminar cualquier dato que pueda contener la tubería de *ftrace* de ejecuciones anteriores.

```
function on_start_test()
{
  echo "ON_START_TEST()"
  sudo timeout 10 cat /sys/kernel/debug/tracing/trace_pipe
}
```

El otro callback, *on_end_test()*, se invoca cuando una carga de trabajo finaliza su ejecución. Su propósito es el de introducir los datos contenidos en la tubería de *ftrace* en el fichero con el resultado de la ejecución generado por Het-Harness, disponible su ubicación en la variable *result_file*. El buffer contiene los datos proporcionados por el módulo del kernel Linux implementado que son de relevancia para realizar evaluaciones sobre la eficacia de algoritmos implementados.

```
function on_end_test()
{
  echo "ON_END_TEST()"
  echo "Writing over ${result_file}"
  sleep 10
  echo "*****************************************************" >> ${result_file}
  echo "BW_SCHED OUTPUT" >> ${result_file}
  sudo cat /sys/kernel/debug/tracing/trace >> ${result_file}
  echo "*****************************************************" >> ${result_file}
}
```

5.1.1.3 Generación de ficheros de cargas de trabajo. Ficheros *.spec

El framework dispone de un script capaz de procesar matrices, donde las filas corresponden a una carga de trabajo y las columnas a los benchmarks que componen cada una de estas, y convertirlas en contenido con un formato válido para poder ser interpretado por el script encargado de lanzar las cargas de trabajo.

Para ilustrar el funcionamiento de este script se presenta el siguiente ejemplo:

1. Se almacena en un fichero de texto, *input_file.csv*, la matriz en el formato descrito al principio de esta subsección:

```
equake00 galgel00 lucas00 swim00 gemsfdtd06 lbm06 leslie3d06 libquantum06
gemsfdtd06 cactusadm06 lbm06 leslie3d06 libquantum06 milc06 soplex06 sphinx306
applu00 galgel00 swim00 lbm06 leslie3d06 libquantum06 milc06 soplex06
```

2. La ejecución del script *runfile2v2* tomando el fichero *input_file.csv* como entrada produce el siguiente resultado. Éste quedará contenido en el fichero *bw_sched_workloads.spec*:

```
$ runfile2v2 input_file.csv > \
$HET_HARNESS_ROOT/benchsets/intel-cat/bw_sched_workloads.spec

#Benchmark List
equake00=(EQUAKE00)
galgel00=(GALGEL00)
lucas00=(LUCAS00)
swim00=(SWIM00)
gemsfdtd06=(GEMSFDTD06)
lbm06=(LBM06)
leslie3d06=(LESLIE3D06)
libquantum06=(LIBQUANTUM06)
cactusadm06=(CACTUSADM06)

...

#Experiments
exp1=(equake00 galgel00 lucas00 swim00 gemsfdtd06 lbm06 leslie3d06 libquantum06)
exp2=(gemsfdtd06 cactusadm06 lbm06 leslie3d06 libquantum06 milc06 soplex06 sphinx306)
exp3=(applu00 galgel00 swim00 lbm06 leslie3d06 libquantum06 milc06 soplex06)
#Test Vector
test_vector=(exp1 exp2 exp3)
```

Puede observarse en el contenido del fichero *bw_sched_benchsets.spec* la relación de benchmarks de los que se hará uso, *Benchmark List*, así como las cargas de trabajo que serán ejecutadas, *expX*, y el orden en el que estas serán procesadas, *test_vector*.

5.1.2 Información proporcionada por Het-Harness

Una vez finalizada la ejecución del script de lanzamiento de experimentos tenemos disponible, por cada carga de trabajo lanzada, un fichero que contiene el resultado de su ejecución.

Het-Harness dispone de un script que tiene como funcionalidad la de procesar individualmente cada uno de esos ficheros para proporcionar al usuario los valores de *slowdown* y de *STP* de la ejecución de cada aplicación contenida en una carga de trabajo determinada.

Es necesario proporcionar al script los tiempos de ejecución de cada uno de los benchmarks cuando estos se ejecutan individualmente en el sistema, *alone_ctimes*. El segundo requisito del script se corresponde con el fichero generado por Het-Harness con la información relativa al resultado de la ejecución de una carga de trabajo en particular, *result_file*.

5.1.2.1 Dependencias. Fichero *alone_ctimes*

Previamente al lanzamiento de cargas de trabajo fue necesario obtener los tiempos de ejecución de la totalidad de los benchmarks, cuando estos se ejecutan individualmente en el sistema, con los que se iba a experimentar. Para ello, se hizo uso de un módulo del kernel Linux, configurado para obtener soporte de PMCTrack [13, 16], ya disponible con la capacidad de hacer asignaciones de último nivel de cache bajo demanda a través de una entrada */proc*.

El fichero está formado por una columna con el nombre del benchmark y con otra que da información acerca del tiempo que llevó su ejecución.

```
ammp00  92859.5411075
applu00 59661.134316
apsi00  82254.587141
art00 24079.2166265
astar06 204534.5767245
bwaves06  664617.405973

[...]
```

5.1.2.2 Salida ofrecida. Fichero *result_file*

Se ilustra, en primera instancia, el contenido del fichero generado por Het-Harness cuando una carga de trabajo finaliza su ejecución.

```
[...]

*** Statistics ***
Completion_time_ms 1850420
*****************
FMA3D00 streams
======
(/tmp/benchlocal/stream05.FMA3D00.err)

real 96.21
user 71.80
sys 0.07
real 112.53
user 71.92
sys 0.10

======
GAP00 streams
======
(/tmp/benchlocal/stream00.GAP00.err)
```

```
real 53.20
user 41.96
sys 0.10
real 52.40
user 40.84
sys 0.09

[...]
```

Por cada una de las ejecuciones de una aplicación en el log queda reflejada la información relativa a su tiempo de procesamiento. Este dato es el que será comparado con el obtenido por cada benchmark reflejado en el fichero *alone_ctimes*.

Contando ya con los requisitos necesarios para obtener la información relativa al valor de *slowdown* y de *STP* de las aplicaciones que componen una carga de trabajo después de finalizar su ejecución [**tbl. 5.1**] se procede a hacer uso del script *parse_result_file_cmp* de tal forma:

```
$ parse_result_file_cmp <alone_ctimes> <result_file>
```

Tabla 5.1: Resumen de la información ofrecida por el framework Het-Harness para una carga de trabajo

Benchmark	ANTT	Slowdown
equake00	0.699282193557	1.4300378434
gap00	0.863866239419	1.15758661974
lucas00	0.76243962976	1.31157925292
swim00	0.837659205962	1.19380291279
gemsfdtd06	0.707933420403	1.41256221444
lbm06	0.734512961215	1.36144636351
leslie3d06	0.886017518044	1.12864585591
libquantum06	0.60262592446	1.65940421646

Mediante esta tabla pueden obtenerse los resultados *slowdown* y de *STP* de una carga de trabajo en particular.

La información ofrecida por Het-Harness será relacionada con el resultado de las ejecuciones producido por la implementación del sistema de logs del módulo del kernel Linux para evaluar la precisión de las estimaciones que se llevan a cabo en este trabajo.

5.2 Evaluación de la implementación

Una vez finalizada la implementación, mediante el uso del framework Het-Harness se lleva a cabo la fase experimental con el fin de verificar la validez de la implementación de los algoritmos diseñados para realizar el cálculo de $BW_{slowdown}$. Dado que este cálculo, aquel que representa la estimación de *slowdown*, se realiza en función del valor obtenido de BW_{alone}, se verificará también la precisión en la estimación de este parámetro respecto al valor real.

Adicionalmente, en esta sección se detalla la forma de obtener cargas de trabajo específicas para llevar a cabo esta evaluación.

Las cargas de trabajo utilizadas en la fase de evaluación están siempre compuestas por 8 aplicaciones coincidiendo con el número de núcleos que dispone el procesador incluido en la plataforma experimental de la que se ha hecho uso en la realización de este trabajo.

5.2.1 Generador de cargas de trabajo

Ya que el propósito final de este trabajo consiste en reducir la métrica *unfairness*, fue implementado un script en Python para encontrar aquellas cargas de trabajo que presentaran un valor más alto de ésta después de simular un número considerable de periodos producidos por el módulo del kernel Linux. En la implementación del generador de cargas de trabajo se utilizó el código del simulador encargado de procesar los datos de entrada relativos al valor medio de BW_{alone} de los benchmarks disponibles, así como el que fue utilizado para simular un número determinado de periodos donde las aplicaciones hacen uso de los recursos del sistema. Además, se implementó la siguiente función:

determine_bw_shared: Dado $bw_{alone}^{\vec{app}}$ y bw_{max} se obtiene $bw_{shared}^{\vec{app}}$ y bw_{total} que corresponde a $\sum_{i=0}^{workload_size} bw_{shared}^{i}$.

Una vez procesados los datos de entrada se hace uso de la librería *itertools* para obtener combinaciones de m elementos tomados de n en n. Concretamente, m es el número de benchmarks disponibles y n el número de elementos que formarán una carga de trabajo coincidiendo con el número de núcleos del procesador disponible.

```python
def generate(self, workload_size=8, bw_max=63580, iterations=1000):
    for workload in itertools.combinations(self.candidates, workload_size):
        appnames = []
        bw_alone_vector = []
        candidates_slowdown = []

        for app in workload:
            appnames.append(app[0])
            bw_alone_vector.append(float(app[1]))

        bw_total, bw_shared_vector = determine_bw_shared(bw_alone_vector, bw_max)

        for i in range(0, workload_size):
            candidates_slowdown.append(float(bw_alone_vector[i]),
                / float(bw_shared_vector[i]))

        r = calculate_unfairness(candidates_slowdown, iterations)
        self.chosen.append((appnames, r))

    self.chosen.sort(key=lambda x: x[1], reverse=True)
```

Procesadas todas las combinaciones de cargas de trabajo posibles y calculados sus valores de *unfairness* se procede a ordenar las resultantes de forma ascendente con la métrica *unfairness* como clave. Se realizará una selección de aquellas cargas que se encuentren en las primeras posiciones favoreciendo la diversidad de aplicaciones que las componen.

5.2.2 Procesamiento de resultados

Con el fin de verificar que las estimaciones consideradas para llevar a cabo la asignación de ancho de banda entre los procesos en ejecución para reducir la métrica *unfairness*, se antojaba necesario escribir un script capaz de procesar el fichero *result_file* de Het-Harness donde se incluye también la información suministrada por el sistema de logs implementado con *ftrace*.

Por simplicidad este script fue escrito en Python. Su comportamiento está basado en buscar todas las filas con la etiqueta *EOE*, que se encuentren en el fichero que contiene el resultado final del lanzamiento de una carga de trabajo, y hacer una clasificación de la información contenida en ellas en función de la aplicación que ha finalizado su ejecución.

La información que ofrece el script es la siguiente [**tbl. 5.2**]:

Tabla 5.2: Resumen de la información ofrecida por el módulo del kernel Linux para una carga de trabajo

APP	SLOWDOWN	ALONE	SHARED	WAYS	ACTIVE	THROTTLED
equake_base.lin	1.434760	8268,087	6307.2830	2.43311	670	16
galgel_base.lin	1.398003	2078.602	1187.3264	2.06785	1663	297
gemsfdtd_base.l	1.402523	5144.349	2830.3095	2.32033	18905	1266
lbm_base.linux-	1.478842	11580.29	9204.5693	2.45379	8569	106
leslie3d_base.l	1.290663	4219.029	3369.6817	2.27869	14129	1139
libquantum_base	1.542162	9952.145	7889.5469	2.48791	11392	3
lucas_base.linu	1.415780	5201.214	3640.0732	2.38916	1240	62
swim_base.linux	1.368616	8989.076	5668.5330	2.51726	1568	10

Donde, por aplicación, se obtiene:

- **SLOWDOWN** es la estimación del valor de *slowdown*; concretamente el valor de $BW_{slowdown}$ acumulado al final de la ejecución.
- **ALONE** corresponde a la media aritmética del cálculo de BW_{alone} realizado en cada periodo en MB/s.
- **SHARED** informa de la media de ancho de banda consumido en MB/s, BW_{shared}, durante la ejecución de la aplicación.
- **WAYS** contiene el valor medio de vías del último nivel de cache que la aplicación dispuso durante su ejecución.
- **ACTIVE** indica cuántas muestras de los contadores hardware fueron tomadas por el módulo del kernel Linux durante la ejecución de la tarea.
- **THROTTLED** esta columna indica cuántas muestras no se tomaron ya que la aplicación se encontraba detenida.

Con estos datos es posible evaluar, en mayor o en menor medida, la validez de la estimación del valor de *slowdown*, el cálculo de $BW_{slowdown}$, y del valor medio de BW_{alone} para cada aplicación conociendo los datos reales, en ambos casos.

El siguiente paso, una vez obtenidos los resultados reales y los ofrecidos por las estimaciones, es relacionar estos para llevar a cabo la fase analítica con el fin de llegar a conclusiones que puedan conducir a realizar modificaciones sobre la implementación.

5.2.3 Evaluación de los resultados

En primer lugar, se llevó a cabo la evaluación de las estimaciones realizadas, de la estimación de *slowdown* y del cálculo de BW_{alone}. Para ello, se lanzaron cargas de trabajo obtenidas mediante el generador de cargas.

Cada aplicación de referencia se encontraba en 4 cargas de trabajo distintas junto con otros 7 benchmarks siendo estos últimos conjuntos ligeramente distintos unos de otros, respectivamente en cada una. De esta forma se pretende evaluar que las estimaciones pueden ser precisas independientemente de la utilización de los recursos del sistema.

Durante esta etapa de evaluación se tomó un periodo de planificación de 10 segundos y un periodo de toma de muestras de 1 segundo.

5.2.3.1 Evaluación del cálculo de BW_{alone} en entorno real

Dado que el cálculo de $BW_{slowdown}$ se realiza en función del parámetro BW_{alone} se antoja necesario verificar, en la medida de lo posible, que éste se aproxima al valor real.

Del mismo modo que se obtuvieron los datos relativos al tiempo de ejecución de todas las aplicaciones que componen el SPEC CPU 2000 y el SPEC CPU 2006 cuando estas se ejecutan individualmente en el sistema, también se obtuvo el valor medio de BW_{alone} real de todas estas en la misma circunstancia.

Lanzada y finalizada la ejecución de todas las cargas de trabajo preparadas para esta evaluación y procesadas mediante scripts de los que ya se ha hecho mención, los datos fueron analizados.

Los resultados de la evaluación llegaron a ser satisfactorios en la mayoría de los casos [**tbl. 5.3**].

- **Benchmark**: Aplicación ejecutada.
- **REAL**: Valor medio de ancho de banda que consumiría la aplicación en caso de no existir competencia por el uso de los recursos compartidos.
- **EST.1...4**: Valor medio de BW_{alone} estimado durante su enésima ejecución.

Los valores de BW_{alone} se exponen en MB/s y corresponden a la media aritmética de todas las estimaciones de este parámetro producidas durante la etapa de ejecución de una aplicación en una determinada carga de trabajo.

En esta tabla se muestra un subconjunto de aplicaciones que pudieron satisfacer la igualdad entre el valor, medio, real y el ofrecido producto de las estimaciones realizadas. Puede verse que el margen de error cometido es asumible.

Tabla 5.3: Estimación de BW_{alone} precisa

Benchmark	REAL	EST.1	EST.2	EST.3	EST.4
fma3d	3684,75843778786	3772,134022	3867,391202	3345,403069	3877,268763
gap	2558,82223754011	2428,889405	2501,330167	2474,161052	2462,607757
lbm	11738,1968260387	12051,535099	11525,263371	11416,091661	11642,287928
leslie3d	4291,49529788152	4366,337619	4200,625077	4264,100008	4525,342027

Sin embargo, en otros casos [**tbl. 5.4**] la estimación quedaba por debajo del valor medio real.

Tabla 5.4: Estimación de BW_{alone} imprecisa

Benchmark	REAL	EST.1	EST.2	EST.3	EST.4
libquantum	12508,3177368392	9365,83392	9333,742613	8618,773733	9644,186712
lucas	5773,30891631098	5000,220977	4833,557494	4953,539659	4693,085897
milc	6259,18236536816	4256,838182	3907,805313	4247,89729	4599,523957
soplex	5193,30405838913	3922,075806	4001,174851	4246,319238	4188,496945

Además de realizar la evaluación sobre la media aritmética del valor de BW_{alone} real y el estimado pudo obtenerse, a través de un script escrito en Python que valida los cálculos que el módulo del kernel Linux lleva a cabo, el valor estimado a lo largo del tiempo, para cada aplicación, cuando cada una de estas se ejecutaba simultáneamente con conjuntos de aplicaciones ligeramente distintos entre sí.

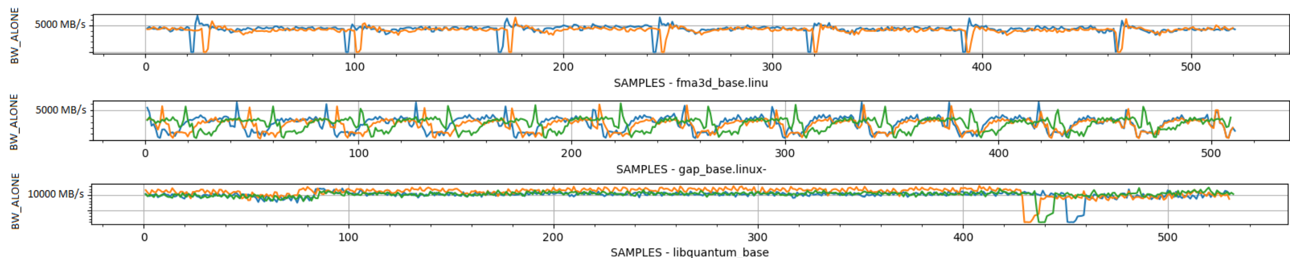


Figura 5.1: Estimación BW_{alone} en entorno real

Con esta gráfica [**fig. 5.1**] pretende ilustrarse que las estimaciones producen resultados similares en cada fase por la que pasa cada aplicación independientemente del contexto en el que se ejecute. Cada una de las líneas representa la estimación del valor de BW_{alone} para una aplicación en una determinada carga de trabajo. El desfase que se produce en cada ejecución se debe a que cada aplicación es perjudicada por la competencia por los recursos compartidos de forma distinta entendiendo que cada instancia se ejecuta en un escenario diferente.

Dado que era posible llevar a cabo estimaciones cercanas a la realidad en una gran selección de aplicaciones, pudo considerarse que la aplicación de las ecuaciones para la estimación de BW_{alone} era satisfactoria para el propósito de este trabajo.

Relevante es que, aún a riesgo de realizar una estimación menos precisa sobre este parámetro, como sucede en algunos casos, no debe descartarse que el cálculo de $BW_{slowdown}$ sea equivalente al valor de *slowdown* real dado que los resultados obtenidos, aunque por debajo de la realidad, son similares entre ellos ya que la degradación que se produce en el rendimiento puede darse por distintos factores.

5.2.3.2 Evaluación del cálculo de $BW_{slowdown}$ en entorno real

Los resultados de esta evaluación, relativos a la verificación de la equivalencia entre $BW_{slowdown}$ y *slowdown*, son los obtenidos después de precisar el cálculo de la métrica BW_{alone} respecto a su valor real. Se utilizaron las mismas cargas de trabajo que en la evaluación anterior, aquellas obtenidas por el generador de cargas.

Se muestra como ejemplo esta figura [**fig. 5.2**] donde se refleja la existencia de buenas estimaciones sobre el *slowdown* al mismo tiempo que se ilustra que el error cometido en la estimación de la degradación en otras aplicaciones es superior a lo que podría considerarse como asumible en esta evaluación.

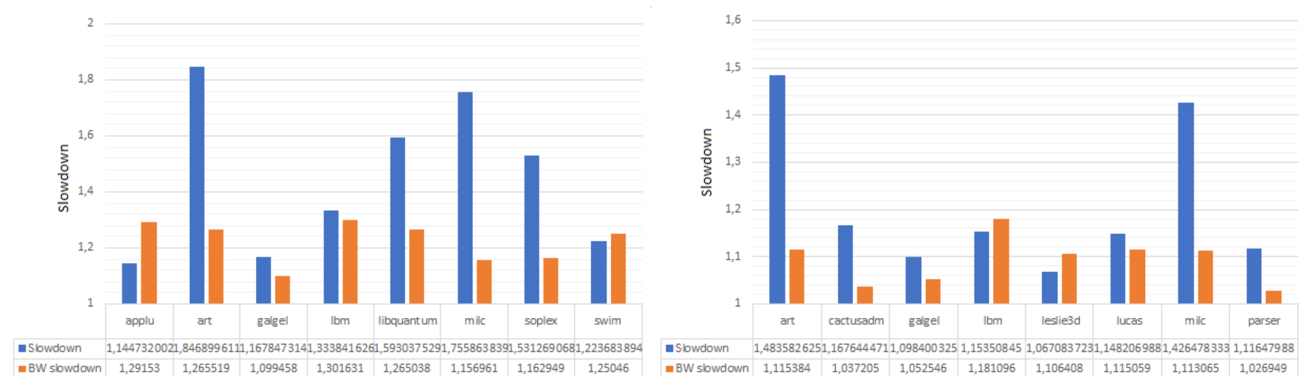


Figura 5.2: Estimación *slowdown* en entorno real - Aplicaciones que maximizan la métrica *unfairness*

Se realizó un estudio acerca del uso de ancho de banda disponible que hacen las aplicaciones en función de las vías de cache disponibles. Dado que el sistema proporciona este recurso de manera equitativa se quiso observar concretamente esta cantidad cuando se dispone de 2 y de 3 vías, de las 20 disponibles. Se pueden hacer 3 observaciones.

1. Independientemente del número de vías de último nivel de cache asignadas a una aplicación esta consume una cuota de ancho de banda equivalente [**fig. 5.3**].

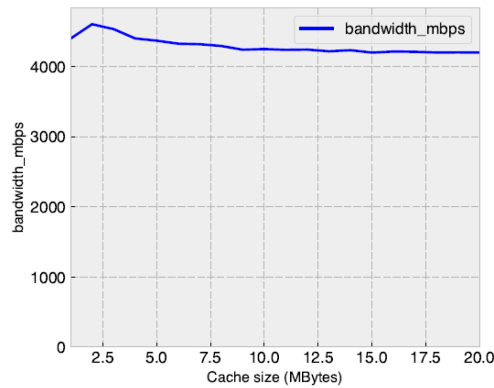


Figura 5.3: Consumo de ancho de banda GemsFDTD - Independencia de la disponibilidad de cache

2. Otro caso se presenta dependiente de la disponibilidad de cache; puede observarse la variación de requisito de ancho de banda siendo éste muy irregular [**fig. 5.4**].

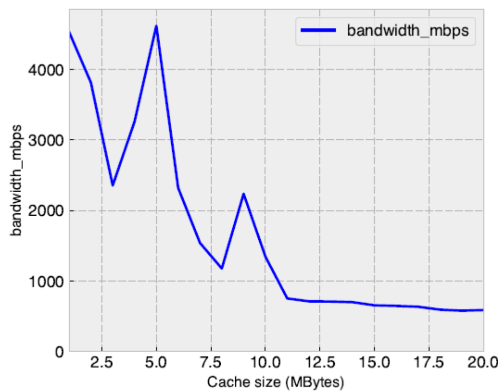


Figura 5.4: Consumo de ancho de banda mcf - Uso irregular en función de disponibilidad de cache

3. Un tercer grupo de aplicaciones dejan de consumir ancho de banda en el momento que se les proporciona un número de vías de último nivel de cache determinado, al contrario que en los casos anteriores [**fig. 5.5**].

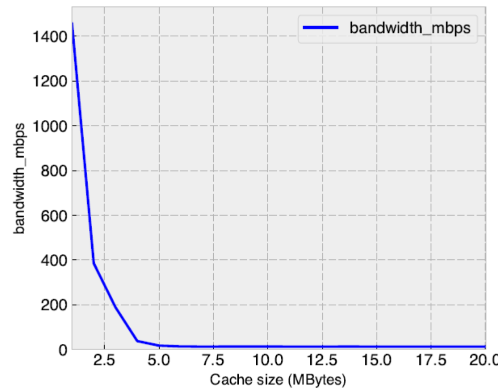
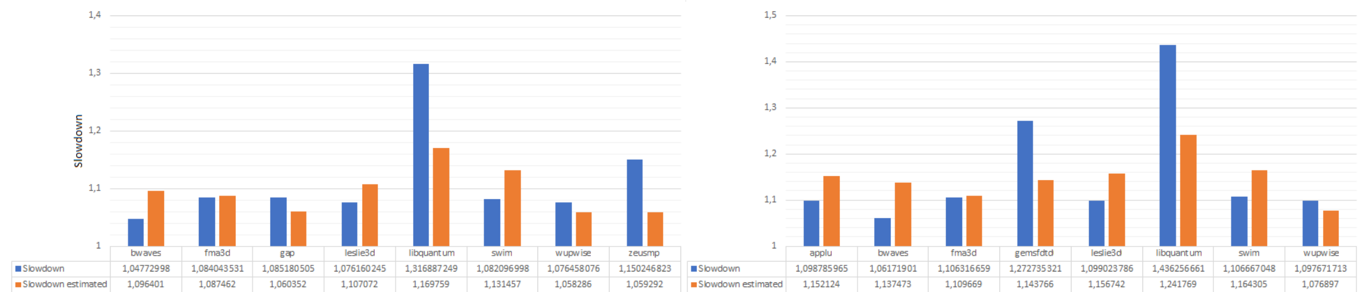


Figura 5.5: Consumo de ancho de banda bzip2 - Diferencia relevante de requisito de ancho de banda en función de disponibilidad de cache

Conociendo el uso que harían las aplicaciones de la memoria principal pudo apreciarse que en aquellas donde se realizaba un consumo de ancho de banda equivalente independientemente de la cache disponible, la estimación realizada contaba con los menores valores de error cometido.

Se lanzaron, por tanto, cargas de trabajo que contenían estas aplicaciones del primer grupo y se realizó una nueva evaluación de la estimación de *slowdown* [**fig. 5.6**].



	bwaves	fma3d	gap	leslie3d	libquantum	swim	wupwise	zeusmp
Slowdown	1,04772998	1,084043531	1,085180505	1,076160245	1,316887249	1,082096998	1,076458076	1,150246823
Slowdown estimated	1,096401	1,087462	1,060352	1,107072	1,169759	1,131457	1,058286	1,059292

	applu	bwaves	fma3d	gemsfdtd	leslie3d	libquantum	swim	wupwise
Slowdown	1,098785965	1,06171901	1,106316659	1,272735321	1,099023786	1,436256661	1,106667048	1,097671713
Slowdown estimated	1,152124	1,137473	1,109669	1,143766	1,156742	1,241769	1,164305	1,076897

Figura 5.6: Estimación *slowdown* en entorno real - Aplicaciones con un consumo de ancho de banda independiente de la cache de último nivel que tengan disponible

Puede observarse que el error cometido es mínimo en muchas de las aplicaciones de esta imagen [**fig. 5.6**], siendo este error equivalente en otras ejecuciones de éstas cuando son ejecutadas en escenarios distintos. Sin embargo, la estimación seguía siendo mejorable en otros benchmarks clasificados en el mismo grupo.

Para alcanzar el reto de reducir la métrica *unfairness* es imprescindible realizar estimaciones precisas sobre la degradación que sufren los procesos en ejecución. Con la intención de que el cálculo de $BW_{slowdown}$ se aproxime lo máximo posible al valor real de *slowdown* en cualquier escenario que pueda presentarse se exploraron alternativas para hacer posible esta equivalencia.

5.3 Implementación de técnicas alternativas para la estimación de la degradación

Ya que resulta imprescindible realizar una buena estimación sobre la degradación que sufren las aplicaciones para tomar decisiones que afecten al uso que deben de hacer éstas de los recursos disponibles para reducir la métrica *unfairness*, en esta sección se expone una nueva definición de $BW_{slowdown}$ así como los métodos empleados para llevar a cabo su implementación.

5.3.1 Nueva definición de $BW_{slowdown}$

Tomando la idea propuesta por otros autores [6] para poder estimar la degradación que sufren las aplicaciones al tener que competir por los recursos del sistema, se planteó hacer uso del modelo estadístico de regresión para realizar una estimación más precisa sobre el *slowdown*. Se toma, por tanto, la siguiente ecuación:

$BW_{slowdown} = \alpha + \beta * \frac{BW_{alone}}{BW_{shared}}$

Aprovechando la capacidad que tiene PMCTrack [13, 16] para obtener métricas relativas a eventos hardware se tomaron aquellas que tienen lugar en el último nivel de cache y en la memoria principal. Se hace empleo de éstas para poder realizar estimaciones sobre los coeficientes α y β, presentes en la nueva definición de $BW_{slowdown}$. Se tuvieron en consideración las siguientes métricas:

- **LLCMPKI**: Relaciona los fallos producidos en el último nivel de cache con el número de instrucciones ejecutadas para obtenerlos.
- **LLCRPKI**: Análoga a la métrica anterior salvo que mide el número de aciertos.
- **LLC_USAGE_KB**: Requisito de ancho de banda reclamado por el último nivel de cache.
- **ROB_STALLS**: Porcentaje de ciclos de parada producidos por accesos al buffer de reordenación cuando éste se encuentra completo.
- **STORE_STALLS**: Porcentaje de ciclos de parada producidos por accesos a los *store buffers* cuando éstos se encuentran completos.
- **MEM_STALLS**: Porcentaje de ciclos de parada producidos por accesos a memoria principal.

MEM_STALLS es una métrica obtenida de manera indirecta a través del contador hardware MEM_LOAD_UOPS_RETIRED.L3_MISS_PS que informa de los fallos en el último nivel de cache y mediante una ecuación proporcionada por el fabricante [28].

$MEM_STALLS = \frac{180 * MEM_LOAD_UOPS_RETIRED.L3_MISS_PS}{CPU_CLK_UNHALTED.THREAD}$

Además de medir el total de ancho de banda consumido pudo diferenciarse, en la medida de lo posible, el motivo por el cuál este recurso fue requerido.

- **BW_LLC_MISSES**: Indica los accesos a memoria con motivo de fallo de último nivel de cache además de los producidos por el mecanismo de *prefetching*; mecanismo implementado en la memoria cache encargado de buscar, de manera anticipada, datos de la memoria principal con el fin de maximizar el número de aciertos en el futuro.
- **BW_LOAD_MISSES**: Muestra únicamente los accesos a memoria producidos por fallos en el último nivel de cache.

Para poder estimar los coefiecientes α y β es preciso en primera instancia obtener los datos reales de éstos para todas las aplicaciones disponibles.

5.3.1.1 Obtención de coeficientes de regresión

En este paso se replicó el experimento presente en la **sección no. 2.2.2** para obtener los valores de $BW_{slowdown}$ y de *slowdown*, además de las métricas necesarias para el nuevo modelo de $BW_{slowdown}$, de aquellas aplicaciones que hacen uso intensivo de la memoria principal variando en cada ejecución de cada una de éstas la carga de trabajo a la que se somete al sistema.

En primer lugar, los datos obtenidos fueron procesados mediante un script desarrollado en Python con el fin de obtener los valores de los coeficientes de regresión α y β para cada relación de *slowdown* y $BW_{slowdown}$ obtenida. Se utilizó nuevamente el lenguaje de programación Python gracias a las utilidades que proporciona el paquete ScyPy [24] para encontrar estos coeficientes.

```python
def calculate_alpha_beta(csv_file):
    dataframe = pandas.read_csv(csv_file)
    benchmarks = numpy.array(dataframe['BENCHMARK'])
    nr_benchmarks = int(numpy.max(numpy.array(dataframe['NR_AGGRESSOR'])))
    slowdown = numpy.array(dataframe['SLOWDOWN'])
    bw_slowdown = numpy.array(dataframe['BW_SLOWDOWN'])
    slowdown_list = numpy.split(slowdown, nr_benchmarks)
    bw_slowdown_list = numpy.split(bw_slowdown, nr_benchmarks)

    output = {}

    for idx in range(0, nr_benchmarks):
        a, b = numpy.polyfit(bw_slowdown[idx], slowdown[idx], 1)
        r = numpy.corrcoef(bw_slowdown[idx], slowdown[idx])
        output[benchmarks[idx]] = a, b, r

    return output
```

Fue posible obtener de este modo valores de α y de β para cada una de las aplicaciones del conjunto seleccionado. Relevante era construir la recta de regresión y observar el valor que toma el coeficiente de correlación de Pearson.

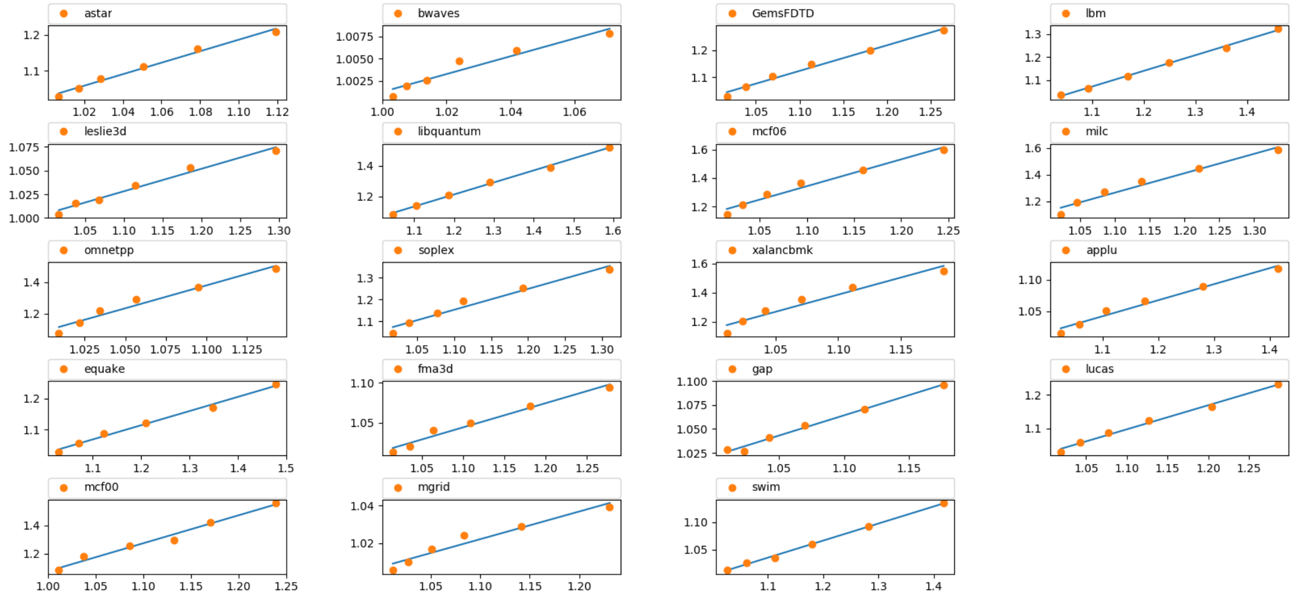


Figura 5.7: Rectas de regresión resultantes de la corrección de la ecuación de $BW_{slowdown}$

Se ilustra el resultado de este experimento [**fig. 5.7**] para todas las aplicaciones con las que se ha llevado a cabo este procedimiento. En todas las figuras el eje de abscisas representa el valor de $BW_{slowdown}$ mientras que el valor de *slowdown* se ubica en el eje de ordenadas. Los puntos representan la equivalencia entre el valor real de *slowdown* y el obtenido mediante $\frac{BW_{alone}}{BW_{shared}}$ mientras que la recta está definida por la ecuación $y = \alpha + \beta * \frac{BW_{alone}}{BW_{shared}}$; en todos los casos el coeficiente de correlación de Pearson es superior a 0.95.

La imagen [**fig. 5.7**] sugiere la posibilidad de hacer uso de la nueva definición de $BW_{slowdown}$ para realizar estimaciones más precisas del valor de *slowdown* de los procesos en ejecución.

5.3.1.1.1 Prueba de concepto en entorno real

En primera instancia, se introdujeron en la implementación de los algoritmos encargados de estimar la degradación los valores de α y de β para cada una de las aplicaciones de manera manual en estructuras de datos. La finalidad era la de verificar que el nuevo modelo propuesto para la estimación de *slowdown* era preciso para que la aplicación de los algoritmos implementados con el fin de reducir el valor de *unfairness* fuese efectiva.

Dado que considerar la nueva ecuación de $BW_{slowdown}$ fue favorable, incluso en el momento que se daba lugar a detener tareas que se encuentran en ejecución cuando éstas habían consumido su cuota de ancho de banda, el siguiente hito era buscar la forma de realizar, en tiempo real, estimaciones para los coeficientes α y β para cualquier aplicación que el módulo del kernel Linux haya capturado para hacer la tarea de seguimiento.

5.3.2 Estimación de coeficientes de regresión

Para estimar los coeficientes α y β en cualquiera de los escenarios que puedan presentarse, teniendo en cuenta las métricas adicionales obtenidas, se utilizó el software Weka [20]; software que se define como una colección de algoritmos de *machine learning* para tareas de minería de datos.

Esta utilidad recibe como entrada una matriz cuyas columnas representan cada una de las métricas disponibles y sus filas los valores asociados a cada una de ellas de cada una de las ejecuciones de todas las aplicaciones seleccionadas. Además de estas métricas, se proporcionaban los valores de *slowdown* y $BW_{slowdown}$ incluyendo los coeficientes α y β obtenidos con anterioridad para cada aplicación.

Se ha de seleccionar qué columnas están disponibles para hacer la estimación y sobre qué columna se debe hacer. Particularmente este procedimiento se realizó dos veces, una para estimar α y otra para hacer lo propio con β no pudiéndose usar uno de los coeficientes para calcular el otro.

En primer lugar, por simplicidad en su posible implementación, se tuvo en cuenta también el uso del modelo de regresión para realizar las estimaciones. Por desgracia el error cometido [**tbl. 5.5**] es muy alto en ambos casos:

Tabla 5.5: Cálculo de coeficientes de regresión mediante regresión

Summary	α	β
Correlation coefficient	0.83	0.8308
Mean absolute error	0.1356	0.1421
Root mean squared error	0.1523	0.1593
Relative absolute error	55.5059%	55.7273%
Root relative squared error	55.7755%	55.6619%

No obstante, la herramienta también se puede configurar para hacer uso del modelo de regresión aditiva; modelo empleado para abordar una problemática similar en trabajos previos [29]. En este caso los resultados [**tbl. 5.6**] de la estimación, tanto como para α como para β fueron excelentes:

Tabla 5.6: Cálculo de coeficientes de regresión mediante regresión aditiva

Summary	α	β
Correlation coefficient	0.9996	0.9996
Mean absolute error	0.0009	0.02
Root mean squared error	0.0022	0.0026
Relative absolute error	1.6709%	3.1691%
Root relative squared error	3.0593%	3.2149%

Además de este resumen, Weka [20] proporciona las operaciones que hay que llevar a cabo para alcanzar el valor que deben de tener los coeficientes dependiendo del valor de las métricas obtenidas mediante los contadores hardware. A modo ilustrativo se presenta un pequeño fragmento de este log obtenido para la estimación del coeficiente β:

```
Additive Regression
Initial prediction: 0.9783749999999999
Base classifier weka.classifiers.trees.DecisionStump
10 models generated.

Model number 0 : Decision Stump : Classifications

BW_LOAD_MISSES <= 561.784306 : 0.024196428571428674
BW_LOAD_MISSES > 561.784306 : -0.16937499999999983
BW_LOAD_MISSES is missing : 1.1102230246251565E-16

Model number 1 : Decision Stump : Classifications

ROB_STALLS <= 0.013600500000000001 : -0.06857142857142856
ROB_STALLS > 0.013600500000000001 : 0.022857142857142854
ROB_STALLS is missing : 0.0

[...]
```

Dada una predicción inicial se va evaluando el valor de las distintas métricas; se ha de acumular al valor inicial un nuevo valor constante dependiendo de si el resultado de la métrica a evaluar se encuentra por encima o por debajo de un umbral.

Considerando en este punto que la corrección de la ecuación de $BW_{slowdown}$ ha resultado ser válida sometiéndola a evaluación en un entorno real, con coeficientes asignados manualmente para cada aplicación, al mismo tiempo que es posible realizar una estimación de éstos con un margen de error mínimo, se procederá a implementar el algoritmo resultado de aplicar el modelo de regresión aditiva para realizar estas estimaciones.

5.3.3 Implementación en el módulo del kernel Linux

Para implementar el algoritmo que permitirá estimar en tiempo real los coeficientes α y β para cualquier aplicación que sea gestionada por el módulo del kernel Linux se siguieron los siguientes pasos:

1. Consultar en el manual del procesador con el que se trabaja los eventos hardware disponibles [30] para obtener los códigos necesarios para establecerlos en la configuración de PMCTrack [13, 16]. Cada vez que se tome una muestra de los contadores hardware esta información estará disponible. Los eventos que serán capturados son aquellos citados al principio de esta sección.

2. PMCTrack [13, 16] tiene soporte para implementar algoritmos que son producto de aplicar el modelo de regresión aditiva. Lo único que se precisa proporcionar son los valores de las métricas requeridas y una estructura de datos con la información proporcionada por el software Weka [20] que representa la toma de decisiones. Este paso se realiza en dos ocasiones; para estimar α por un lado y β por otro.

3. Por último, los valores estimados de los coeficientes son obtenidos además del valor de ancho de banda consumido por la aplicación sobre la que se recoge la muestra. Requisitos necesarios para el nuevo cálculo de $BW_{slowdown}$.

A modo ilustrativo se muestra un fragmento de la estructura de datos requerida para poder llevar a cabo las estimaciones de los coeficientes de regresión. Concretamente, ésta es la utilizada para realizar la estimación del coeficiente β:

```
int additive_regression_b_model[]= {
    978,101,954,-1,
    BW_LOAD_MISSES_METRIC,561784306,24,-169,
    ROB_STALLS_METRIC,13600,-68,22,
    [...]
    -1,-1,-1,-1
};
```

En la primera fila se muestra la predicción inicial, el mínimo y el máximo valor obtenido del coeficiente a estimar en el momento que se obtuvieron todos los valores de este coeficiente para todas las muestras que recibió Weka [20] como entrada. Las filas sucesivas están compuestas por la posición que ocupa la métrica en el vector donde se almacenan todas ellas, el valor del umbral y las constantes que hay que sumar a la predicción inicial en función del valor de la métrica en cuestión respecto a su umbral. Es importante que estas filas se declaren en el mismo orden que la sucesión de modelos propuestos por el software de minería de datos.

Adicionalmente se expone el algoritmo encargado de llevar a cabo la toma de decisiones:

```
int estimate_sf_additive(uint64_t* metrics, int* adregression_spec)
{
    int sf = adregression_spec[0];
    int sfmin = adregression_spec[1];
    int sfmax = adregression_spec[2];
    int idx = 4;
    int metric_id;

    while(adregression_spec[idx] != -1) {
        metric_id = adregression_spec[idx];
        if (metrics[metric_id] <= adregression_spec[idx+1]) {
            sf += adregression_spec[idx+2];
        } else {
            sf += adregression_spec[idx+3];
        }
        idx += 4;
    }

    if (sf < sfmin) {
        sf = sfmin;
    } else if (sf > sfmax) {
        sf = sfmax;
    }

    return sf;
}
```

Es imprescindible proporcionar a la función *estimate_sf_additive* todos los valores de las métricas que han de considerarse para realizar la estimación de un coeficiente en particular, *metrics*, además de la estructura de datos que representa la toma de decisiones, *adregression_spec*. Se observa que el bucle principal evalúa cada una de las métricas y añade a la predicción inicial el valor constante que procede después de comprobar si el valor de la enésima métrica del vector se encuentra por encima o por debajo del umbral establecido para ésta.

Una vez finalizada esta implementación se procede a evaluar si la nueva definición de $BW_{slowdown}$ se aproxima en mayor medida al valor real de *slowdown* en cualquiera de los contextos que puedan presentarse permitiendo así evaluar los algoritmos diseñados para reducir la métrica *unfairness*.

5.4 Resultados finales

Se presentan en esta sección las conclusiones finales de la evaluación de las estimaciones consideradas, concretamente de la estimación de *slowdown* después de explorar nuevas posibilidades para que el cálculo de $BW_{slowdown}$ se aproxime lo máximo posible. A continuación, se pondrá a examen el contexto en el que esta estimación permite reducir el valor de la métrica *unfairness* obtenida cuando la ejecución de un conjunto de tareas llega a su fin.

5.4.1 Minimizando la métrica unfairness

Después de hacer uso del nuevo método para estimar el valor de *slowdown* sobre las aplicaciones del SPEC CPU 2000 y del SPEC CPU 2006, los resultados fueron satisfactorios. En general, se observó que los cálculos de $BW_{slowdown}$ se aproximaban aún más a los respectivos valores reales de *slowdown* en todos los escenarios que se evaluaron. Dado que ahora sí existe la posibilidad de hacer buenas estimaciones sobre la degradación que sufren las aplicaciones pudieron refinarse aspectos relacionados con la asignación de los recursos disponibles.

Finalmente, se han respetado las ideas propuestas en este trabajo cuando, puestas en práctica en el contexto real, han resultado ser satisfactorias. No obstante, definir cuál sería el valor de $BW_{slowdown}$ que debe alcanzar la aplicación que maximiza este parámetro es aún una tarea pendiente. El establecimiento de un valor constante no podía considerarse debido a que éste debía estar relacionado con el progreso que tienen las aplicaciones en un momento dado; no se establece de tal manera debido a que los resultados experimentales evidenciaban la necesidad de que éste se modificara en función de la carga de trabajo presente.

Se tomó en primer lugar una idea [6] basada en aumentar o disminuir el valor de $slowdown_target$ considerando el número de aplicaciones en ejecución que tienen un valor de $BW_{slowdown}$ por encima o por debajo respectivamente de él; el establecimiento inicial consistía en encontrar un valor que separase en conjuntos con el mismo cardinal a los procesos en ejecución. En la consecución de cada periodo se tomaban datos para evaluar, a corto plazo, si el valor de $slowdown_target$ podría considerarse inalcanzable para aumentarlo ligeramente en ese caso o disminuirlo en caso de detectar tendencia a alcanzarlo. Esta propuesta quedó descartada porque, generalmente, no ofrecía los resultados esperados; en el escenario donde se desiste que la aplicación con mayor valor de $BW_{slowdown}$ alcance el $slowdown_target$ establecido por considerarse inalcanzable, con anterioridad a aumentar su valor se imponen penalizaciones mayores sobre aplicaciones que, de haber intentado alcanzar un objetivo más próximo, podrían incluso haber sido consideradas para favorecerse de las situaciones de baja contención generadas.

Ha de tenerse en cuenta que imponer penalizaciones a algunas aplicaciones para que otras vean mejorado su rendimiento aumenta el valor de *slowdown* de las primeras, por lo que, con el paso del tiempo este objetivo debe ser cada vez más austero por la tendencia que existirá a que los valores de $BW_{slowdown}$ de los procesos en ejecución se aproximen entre sí.

Tomando esto en consideración se escribió un algoritmo que ordena, de manera descendente, un conjunto de aplicaciones en función de su valor de $BW_{slowdown}$ para posteriormente detectar qué aplicación presenta mayor diferencia de este parámetro con la que le sigue en la lista; en caso de detectar esta diferencia en varias ocasiones se tiene en cuenta la primera vez que se encuentra. En ese momento:

1. Las aplicaciones se dividen en dos grupos, las que están por encima y las que están por debajo de aquellas donde se encuentra la mayor diferencia; las aplicaciones de referencia son repartidas entre los dos conjuntos. Cabe destacar que esta diferencia debe superar un umbral establecido en función de las aplicaciones que maximizan y minimizan, respectivamente, el valor de $BW_{slowdown}$; si éste no es superado no se llevará a cabo ninguna planificación al entender que el progreso de las tareas en ejecución es similar en todas ellas.
2. Se obtiene la media aritmética de los valores de $BW_{slowdown}$ del grupo de aplicaciones que minimizan este valor. Éste será el valor de $slowdown_target$.
3. Se aplica la ecuación que da como resultado el ancho de banda a repartir entre todas las aplicaciones para que la de referencia alcance el objetivo marcado. Se considera que la aplicación de referencia, a partir de ahora, es la que minimiza el valor de $BW_{slowdown}$ en el grupo de aplicaciones que maximiza este parámetro.
4. La cuota de ancho de banda a repartir es considerada en primer lugar para asegurar que las aplicaciones que maximizan el valor de $BW_{slowdown}$ no puedan ser detenidas. El ancho de banda restante es asignado entre las aplicaciones del segundo grupo con el algoritmo encargado de hacer el reparto de este recurso en función del peso que tenga cada aplicación.

5.4.1.1 Resultados experimentales

En esta subsección se analizará con alto nivel de detalle los resultados ofrecidos por los algoritmos implementados.

La configuración del módulo del kernel Linux [**tbl. 5.7**] que ha ofrecido los mejores resultados es la siguiente:

Tabla 5.7: Configuración final del módulo del kernel Linux

Parámetro	Valor
Periodo de toma de muestras	50 milisegundos
Periodo de planificación	500 milisegundos
Técnica para reducir la contención	Detención de tareas

5.4.1.1.1 Fase 1 - Prueba de concepto

En primer lugar, debía poder confirmarse que las detenciones planificadas podían conducir a que las aplicaciones que maximizaran el valor de $BW_{slowdown}$ pudieran favorecerse de la reducción del consumo de cuota de ancho de banda a nivel global durante un determinado periodo de tiempo.

Para el primer experimento se tomó el benchmark libquantum como aplicación de referencia debido al alto requisito de accesos a memoria que tiene durante toda su ejecución respecto al resto. La elección de las aplicaciones que se ejecutarían simultáneamente con ésta se basa en el consumo de ancho de banda medio de cada una de ellas con el fin de someter al sistema a cargas de trabajo distintas para validar la eficacia de la implementación en función de los distintos grados de contención que pueden darse en los accesos a memoria principal.

Se presenta en la figura posterior a los siguientes párrafos [**fig. 5.8**] 3 cargas de trabajo numeradas al mismo tiempo que se ilustra el consumo de ancho de banda medio de las 3; en ésta última se muestra una columna con el volumen de ancho de banda que consumirían todos los procesos de la respectiva carga de trabajo en caso de no existir competencia por los recursos, **TEÓRICO**, y el valor medio medido, **MEDIDO**, del ancho de banda total consumido durante la ejecución. Puede observarse que existe la posibilidad de que aplicaciones que se encuentren en ejecución que maximicen el valor de *slowdown* puedan llegar a beneficiarse de la detención de otras tareas que sufren en menor medida esta degradación.

La reducción de la métrica *unfairness* pudo darse en estos distintos contextos [**tbl. 5.8**] siendo siempre esta reducción superior a la pérdida de rendimiento [**tbl. 5.9**]. La optimización de justicia y del rendimiento global son típicamente objetivos contrapuestos; intentar mejorar uno de los objetivos suele afectar negativamente al otro [31, 29, 32].

BASE: Resultados sin llevar a cabo ninguna planificación. **SCHED**: Resultados cuando se aplican los algoritmos encargados de tratar de reducir la métrica *unfairness*.
DIFF %: Porcentaje de diferencia entre BASE y SCHED.

Tabla 5.8: Reducción de *unfairness* - Prueba de concepto

Workload	UNFAIRNESS BASE	UNFAIRNESS SCHED	UNFAIRNESS DIFF %
Workload #1	1,47026121	1,24940773	15,02%
Workload #2	1,38214574	1,16424961	12,40%
Workload #3	1,24834893	1,12536629	9,85%

Tabla 5.9: Reducción de *throughput* - Prueba de concepto

Workload	THROUGHPUT BASE	THROUGHPUT SCHED	THROUGHPUT DIFF %
Workload #1	6,09433709	5,56484689	8,69%
Workload #2	6,87100831	6,29017713	8,45%
Workload #3	7,31490801	6,74144401	7,83%

Se observa en el gráfico junto al resultado de la ejecución de las 3 cargas de trabajo [**fig. 5.8**] que, según disminuye el requisito de ancho de banda total, el consumo que hacen de éste se asimila aún más al que harían de no existir competencia por los recursos. Este fenómeno tiene relación también con la posibilidad de acelerar la ejecución de tareas con mayor *slowdown*; puede verse que la aplicación de referencia, libquantum, puede disminuir este factor en menor medida según se reduce la carga de trabajo introducida.

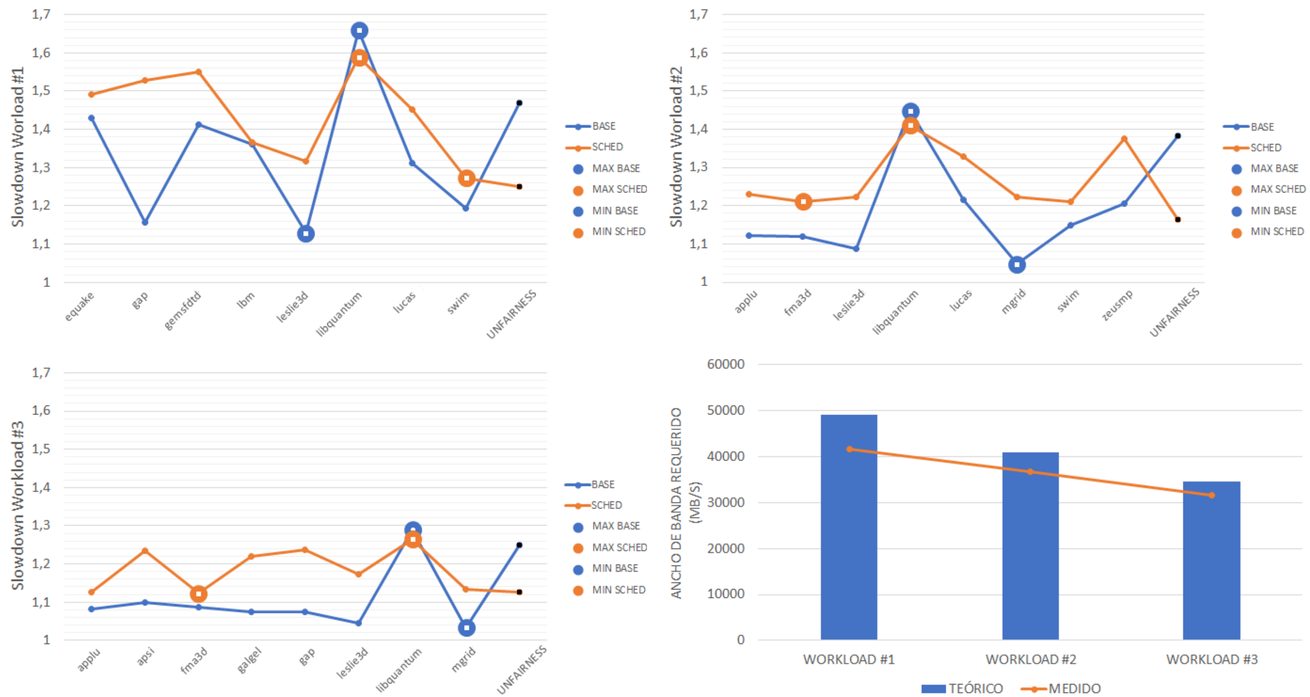


Figura 5.8: Reducción de *unfairness* en función de la carga de trabajo - Prueba de concepto

Siendo consciente de que puede darse la situación en la que las aplicaciones que sufren mayor *slowdown* no obtendrán beneficio de la reducción de consumo de ancho de banda a nivel global al producirse detenciones, la implementación está preparada para no actuar en este contexto.

Se expone en las siguientes figuras [**fig. 5.9**] dos ejemplos que denotan que la planificación no da lugar a asignaciones de ancho de banda. En la primera se escogieron las 8 aplicaciones con menor consumo de ancho de banda, sustituyendo galgel por applu para introducir un grado de contención algo mayor respecto a la otra gráfica, y en la segunda otras tantas donde su rendimiento no está relacionado estrictamente con la cuota de ancho de banda que dispongan. El mecanismo encargado de la detección de este tipo de situaciones toma un papel relevante en este punto; en el segundo caso está más que justificada su existencia, sin embargo, en el primero cabe añadir que se ha observado que la implementación no es capaz de ofrecer aumentos de rendimiento provocando la aparición de intervalos donde la contención en los accesos a memoria principal es menor cuando ésta ya es baja. Este último escenario fue el motivo por el que se decidió no realizar acciones si éste se presenta.

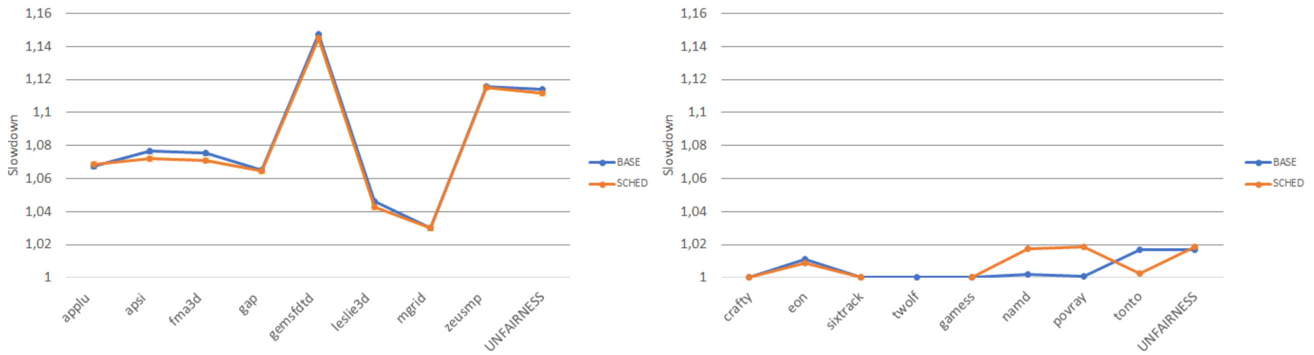


Figura 5.9: Grado de contención en los accesos a memoria principal insuficiente para aplicar los algoritmos encargados de reducir la métrica *unfairness*

En la siguiente figura [**fig. 5.10**] se ilustra el uso de ancho de banda que hace cada aplicación en relación al que haría de no existir competencia por los recursos así como el porcentaje de tiempo que cada aplicación ha sido detenida en esta primera fase experimental. Se puede observar que la implementación penalizará a las aplicaciones donde detecta un valor menor de $BW_{slowdown}$.

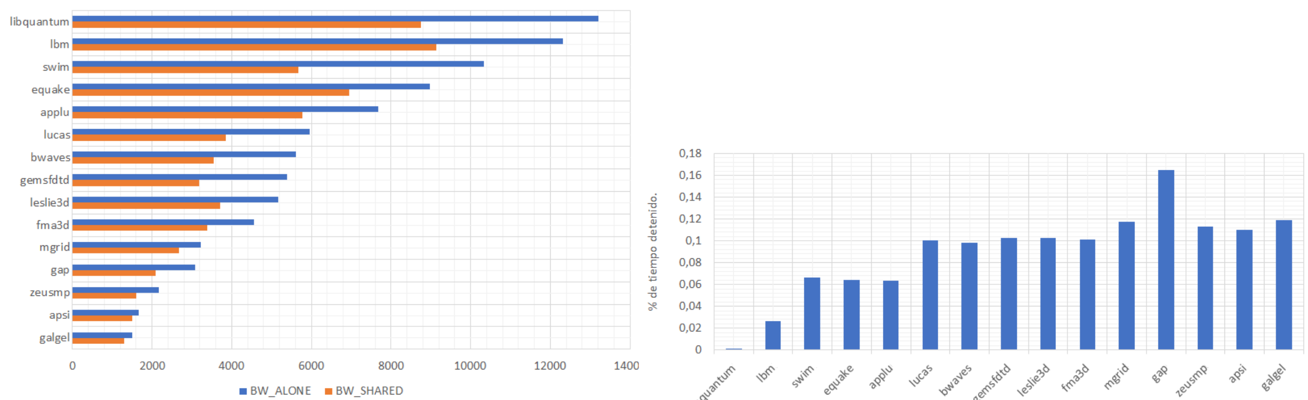


Figura 5.10: Ancho de banda consumido por aplicación y tiempos de desactivación

5.4.1.1.2 Fase 2 - Evaluación de los resultados que ofrece la implementación

Después de analizar los resultados presentados en la primera fase, surge la necesidad de introducir réplicas de las aplicaciones que tienen de media un consumo de ancho de banda mayor respecto al resto para provocar que la degradación que sufren las aplicaciones debido a la competencia por los recursos sea mayor. Se expuso anteriormente la existencia de un algoritmo que trata de distinguir dos conjuntos de aplicaciones en función de su progreso para primar a aquellas que sufren en mayor medida ejecutarse simultáneamente con otros procesos.

Es por ese motivo por lo que en primera instancia se decide replicar la aplicación libquantum. Este benchmark tiene un consumo de ancho de banda muy superior al del resto de aplicaciones durante toda su ejecución; hecho que produce poder distinguir claramente un conjunto de aplicaciones que maximizarán el valor de $BW_{slowdown}$ respecto a otro que a riesgo de sufrir también la competencia por los recursos lo hace en menor medida. Puede verse en la siguiente imagen [**fig. 5.11**] que la implementación es capaz de detectar estos dos conjuntos cuando encontramos 3 instancias de libquantum ejecutándose simultáneamente. Es visible que la asignación de los recursos resulta favorable para estos procesos siendo aumentado el rendimiento en todos los casos.

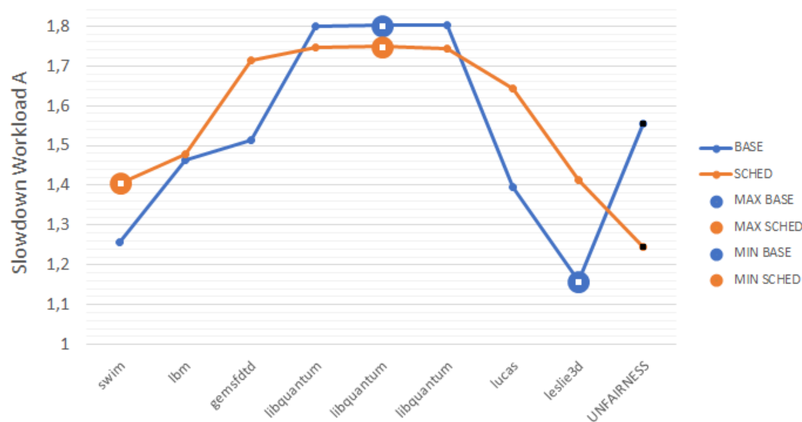


Figura 5.11: Carga de trabajo elevada donde se reduce la métrica *unfairness*

Otro motivo que llevó a utilizar réplicas de aplicaciones era el de tratar de distinguir estos conjuntos cuando los valores de $BW_{slowdown}$ se encuentran próximos entre sí; aunque lo suficientemente distantes para aplicar las operaciones encargadas de tratar de reducir la métrica de *unfairness*.

Con el benchmark libquantum presente en la ejecución de una carga de trabajo distinguir estos dos conjuntos es una tarea sencilla. En muchos experimentos se observaba que la competencia por los recursos perjudicaba también a aplicaciones como lbm, equake y gemsfdtd. Cabe destacar que la penalización que sufrían era notable debido a la existencia de procesos en ejecución con un consumo de ancho de banda por encima o similar al que hacen éstas, por tanto, se decide replicar instancias de estos benchmarks por separado para encarecer los accesos a memoria principal y juntarlas en sus respectivas cargas de trabajo con otras con un consumo algo menor pero que pudiera influir en el avance de éstas.

Nuevamente se tendrá en cuenta la idea de poder distinguir dos conjuntos de aplicaciones en función de su valor de $BW_{slowdown}$.

En el caso anterior [**fig. 5.11**] se tuvo en consideración que las aplicaciones que se escogieran para que la de referencia aumentase su rendimiento tuvieran un valor de *slowdown* lo más distinto posible entre ellas; esto se hizo para verificar que existiría tendencia en tratar de minimizar el valor de *slowdown* de los procesos, y no sólo de uno, que lo maximizasen cuando en una carga de trabajo se encuentran aplicaciones que hacen distinto uso de los recursos disponibles. Para el siguiente experimento, donde se tratará de maximizar el rendimiento de las aplicaciones mencionadas en el párrafo anterior, se escogieron cargas de trabajo atendiendo a los siguientes criterios:

1. Sin llevar a cabo ninguna planificación el intervalo marcado por el valor mínimo y valor máximo de *slowdown* de cada carga de trabajo no debe estar contenido estrictamente en ningún otro.
   - Workload B: Intervalo [1,189, 1,458]
   - Workload C: Intervalo [1.085, 1.331]
   - Workload D: Intervalo [1,025, 1,169]
2. Las aplicaciones pueden replicarse dentro de una carga de trabajo siempre y cuando se cumpla que existan 3 aplicaciones distintas, como mínimo. Lo que no está permitido es el uso de un mismo benchmark en varias cargas de trabajo.
   - Workload B: lbm, swim, fma3d
   - Workload C: equake, leslie3d, lucas
   - Workload D: gemsfdtd, bwaves, mgrid
3. Cada conjunto debe estar formado por aplicaciones donde, en primera instancia, el número de procesos que serán seleccionados para aumentar su rendimiento es distinto respecto al otro.
   - Workload B: Dos conjuntos de aplicaciones que minimizan el valor de *slowdown* (swim, fma3d) están próximos entre sí respecto a un tercer conjunto que lo maximiza (lbm).
   - Workload C: Dos conjuntos de aplicaciones que maximizan el valor de *slowdown* (equake, lucas) están próximos entre sí respecto a un tercer conjunto que lo minimiza (leslie3d).
   - Workload D: Este caso es una réplica del método que se utilizó para elegir las aplicaciones del Workload B con las aplicaciones (bwaves, mgrid) y (gemsfdtd) respectivamente. La justificación de tomar nuevamente esta idea es la de hacer la evaluación de la implementación con una carga de trabajo mucho menor poniendo a prueba la estimación del valor de *slowdown* en este contexto.

Con el punto 1 lo que se trata es de evaluar la implementación en función de las distintas cargas de trabajo a las que se le puede someter al sistema. Dado que el rango de los intervalos es lo más reducido posible se pondrá también a examen la estimación del valor de *slowdown* mediante el cálculo de $BW_{slowdown}$; una buena estimación es imprescindible en este escenario.

El punto 2 simplemente marca unas reglas en cuanto al uso de réplicas de aplicaciones dentro de una misma carga de trabajo y entre las que se realizará esta evaluación. Se han escogido los benchmarks que se encontraban en la posiciones más altas en la figura [**fig. 5.10**], donde se detallaba el uso medio que hacen del ancho de banda durante su ejecución, descartando aquellos que se encuentran en las posiciones inferiores.

Finalmente, el punto 3 muestra los motivos para utilizar las aplicaciones que se han asignado para cada carga en cuestión. Dado que pueden existir varios grupos de aplicaciones en función del *slowdown* que sufren, el propósito de esta selección es verificar si la implementación acaba favoreciendo a los procesos que maximizan este valor.

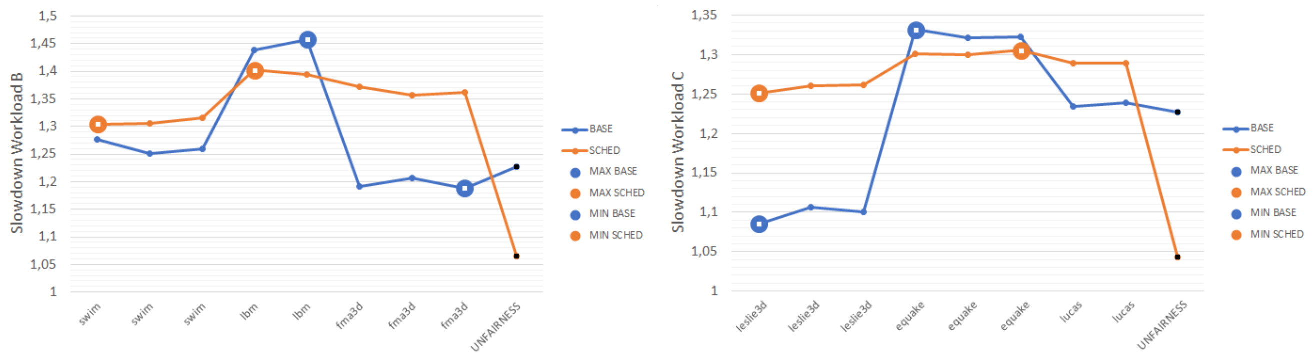


Figura 5.12: Cargas de trabajo diversas donde se reduce la métrica *unfairness*

Esta imagen [**fig. 5.12**] ilustra que es posible aumentar el rendimiento de varias aplicaciones, si se logra hacer correctamente la distinción entre conjuntos en función del valor de $BW_{slowdown}$ de los procesos en ejecución, realizando la asignación de los recursos disponibles propuesta en este trabajo.

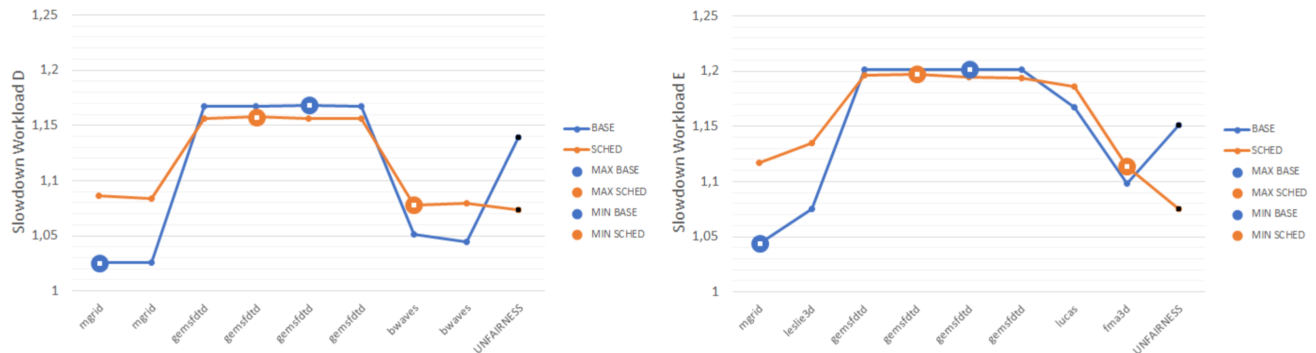


Figura 5.13: Reducción de la métrica *unfairness* en contextos extremos

La figura [**fig. 5.13**] representa el límite sobre el que la implementación es capaz de realizar su cometido con éxito. Adicionalmente a Workload D se incluye una nueva carga, Workload E, donde se han añadido aplicaciones existentes en otras cargas de trabajo para verificar que la estimación del parámetro *slowdown* es precisa en otro contexto distinto que pueda presentarse.

En esta fase, donde se ha tratado de verificar que la implementación encargada de asignar los recursos disponibles haciendo una distinción entre el valor de $BW_{slowdown}$ que tienen las aplicaciones en ejecución, puede verse también que la mejora en el rendimiento que pueden sufrir las aplicaciones más perjudicadas por la competencia por los recursos compartidos tiene relación precisamente con el grado de ésta igual que se observó en la primera fase. Este experimento muestra la tendencia que tiene la planificación llevada a cabo a aproximar el valor de $BW_{slowdown}$ de todas las aplicaciones que se encuentren en ejecución entre sí; tendencia limitada en algunos casos por el error cometido en la estimación de los valores de *slowdown*.

Sin olvidar que el propósito de este trabajo es reducir la métrica *unfairness* se presentan las siguientes tablas para poder mostrar que estos experimentos cumplen dicho objetivo y que, igual que en los casos anteriores, éste ha sido alcanzado siendo el porcentaje de diferencia de la reducción de la métrica *unfairness* mayor [**tbl. 5.10**] que el respectivo al evaluar la pérdida de rendimiento global [**tbl. 5.11**].

Tabla 5.10: Reducción de *unfairness* en cargas de trabajo diversas

Workload	UNFAIRNESS BASE	UNFAIRNESS SCHED	UNFAIRNESS DIFF %
Workload A	1,55449392	1,2456159	19,87%
Workload B	1,22658353	1,07552730	12,31%
Workload C	1,22672771	1,04355817	14,93%
Workload D	1,13929271	1,07391091	5,74%
Workload E	1,15090839	1,07499474	6,59%

Tabla 5.11: Reducción de *throughput* en cargas de trabajo diversas

Workload	THROUGHPUT BASE	THROUGHPUT SCHED	THROUGHPUT DIFF %
Workload A	5,38283633	5,00549401	7,01%
Workload B	6,2682229	5,92308787	5,51%
Workload C	6,61475589	6,24123998	5,65%
Workload D	7,28525038	7,15681071	1,76%
Workload E	6,98454771	6,86400961	1,72%

La pérdida de rendimiento global tiene una causa lógica en este escenario: se interrumpe completamente la ejecución de procesos para que otros puedan progresar una proporción menor de lo que avanzarían las tareas detenidas en caso de no aplicarlas ninguna penalización. Lo que no estaría justificado es que las penalizaciones no impliquen un aumento en el rendimiento de las aplicaciones que sufren mayor *slowdown*. Por ello, se han implementado mecanismos para que esto no suceda. Además de haber obtenido una relación favorable entre la pérdida de rendimiento y la disminución del valor de *unfairness* puede confirmarse también con estos resultados que los algoritmos implementados tienen cabida si se precisa aumentar el rendimiento de aplicaciones que sufren en mayor medida la degradación producida por la competencia por los accesos a memoria principal.

Capítulo 6

Conclusiones y trabajo futuro

6.1 Conclusiones

Durante el desarrollo de este trabajo se ha podido observar el comportamiento que tienen distintas aplicaciones en función de los recursos disponibles. En el contexto de la computación en la nube, garantizar calidad de servicio a todos los usuarios de una plataforma puede suponer un reto debido a que los servidores destinados para este fin ejecutan procesos que pueden hacer uso de los recursos compartidos de manera muy diversa.

Este Trabajo de Fin de Grado ha propuesto un algoritmo de gestión de los recursos compartidos en tiempo real enfocado en que procesos que dispongan de la misma prioridad dentro de un mismo sistema sufran del mismo grado de degradación al hacer uso de estos recursos optimizando así la justicia y favoreciendo la calidad de servicio.

Debido a la disponibilidad de una plataforma experimental que cuenta con las tecnologías Intel CAT [15] e Intel MBM [17] el objetivo ha sido gestionar el uso de los recursos compartidos cuando se están ejecutando simultáneamente en el mismo sistema aplicaciones que hacen un uso intensivo tanto de la memoria cache de último nivel como de la memoria principal. Este escenario fue considerado ya que fue posible estimar la degradación que sufren este tipo de procesos [5] haciendo mediciones sobre el ancho de banda que consumen al mismo tiempo que se obtenía información relativa a métricas de rendimiento procedentes de los contadores hardware.

La herramienta open source de monitorización PMCTrack [13, 16] permite al usuario poder monitorizar estos eventos desde una interfaz de línea de comandos de manera sencilla. Se hizo uso de ésta sobre todo en las primeras fases de este trabajo cuando se trataba de detectar tendencias después de aplicar restricciones a los procesos sobre el uso de los recursos del sistema. Posteriormente, se empleó el soporte de esta herramienta para el desarrollo de un módulo del kernel Linux donde se encuentra el algoritmo de gestión de los recursos compartidos. La principal ventaja que ha proporcionado ha sido la capacidad de abstracción que ha ofrecido al programador para poder considerar eventos obtenidos del hardware para la implementación del algoritmo.

Para evaluar la eficacia del algoritmo propuesto fue utilizado un framework, Het-Harness, que permitió lanzar diversas cargas de trabajo de manera sistemática en la plataforma experimental reportando el resultado de la ejecución de cada una de ellas. Se hizo uso de las suites SPEC CPU 2000 y SPEC CPU 2006 para evaluar la implementación realizada. De este modo, pudieron obtenerse conclusiones relevantes acerca de la implementación.

Los últimos resultados obtenidos habiendo realizado una gestión en tiempo real sobre el uso de los recursos compartidos confirman tendencias significativas de igualar la degradación que sufren procesos en ejecución con la misma prioridad.

6.2 Trabajo futuro

Las conclusiones que se han presentado están estrechamente relacionadas con hacer uso del potencial que tienen las herramientas utilizadas, tanto hardware como software, debido a distintas observaciones presentadas a lo largo de la fase de experimentación.

A lo largo del desarrollo de este proyecto se ha podido observar el comportamiento de las aplicaciones en función de los recursos que les son proporcionados. Haciendo uso de métricas de rendimiento y del soporte de la tecnología Intel MBM [17] ha resultado posible establecer un perfil para cada proceso en ejecución con el fin de priorizar el uso de los recursos compartidos en función de éste. No hay que olvidar que las tareas, durante su ejecución, pasan por distintas fases por lo que es necesario hacer un seguimiento continuo para que la planificación del uso de los recursos compartidos pueda resultar eficiente. Mediante estas métricas también ha sido posible establecer un perfil que relaciona a aquellas aplicaciones que dejan de consumir ancho de banda en el momento que tienen suficiente espacio en el último nivel de cache en contraposición a aquellas que hacían uso del ancho de banda con la memoria principal indistintamente de la disponibilidad de la cache disponible.

- Se propone, por tanto, seguir haciendo uso de la tecnología Intel CAT [15] pero con el fin de priorizar el uso del último nivel de cache a aquellas aplicaciones donde pueda detectarse mayor aprovechamiento de este recurso. Esto aumentará en algunos casos el rendimiento de determinados procesos mientras que aquellos que hacen un uso intensivo de la memoria principal también serán favorecidos por los intervalos de baja contención en los accesos a memoria que puedan generarse como consecuencia.

Gracias a Intel MBM [17] y a los contadores hardware ha sido posible estimar la degradación que sufren las aplicaciones que hacen un uso intensivo de la memoria principal para imponer restricciones en el uso de este recurso. Dado que el hardware utilizado no permite por sí mismo limitar la utilización del ancho de banda disponible se delegó sobre el software esta tarea, lo que supuso no poder precisar el uso que se establece de antemano del ancho de banda disponible para las aplicaciones en ejecución.

- Poder contar con hardware capacitado para imponer estas restricciones por sí mismo en el momento que se detecte que las aplicaciones alcancen el límite establecido puede llegar a favorecer al algoritmo propuesto en este trabajo. Sería muy interesante someterlo a evaluación en este contexto.

En el momento que un proceso no realiza un consumo de ancho de banda que pueda resultar significativo la aplicación del algoritmo propuesto no resulta ser transcendental ya que es incapaz de estimar la degradación en el rendimiento que sufren ese perfil de aplicaciones.

- El estudio de técnicas, basadas en la monitorización de métricas que pudieran ser relevantes, que permitan estimar este fenómeno en el contexto que se presenta y que además puedan ser implementadas en el kernel Linux podría resultar ser de utilidad.

Distinguir estos distintos perfiles de aplicaciones presentados puede llevar a realizar una gestión eficaz de los recursos compartidos favoreciendo la calidad de servicio que una plataforma puede proporcionar.

Appendix A - Introduction

Motivation

Since ‘80s, the speed of processors has been improved by approximately 50% to 100% each year while, in DRAM memories, it has only grown by around 7% every year [1]. The concept ’memory wall’ finds its meaning in this context [2]; in the progressive improvement in the speed of the processors as opposed to the advances in the speed of these memories.

At the beginning of the century, there was a tendency to increase the number of cores in the same processor, increasing even more the demand for main memory bandwidth. Therefore, the importance in the design of the memory hierarchy has grown in function of the advances in the performance of the processors.

Nowadays there are researches focused on increasing the performance of the shared cache memories [3, 4] while other studies are related to decrease the main memory contention. This problem has been addressed from different points of view: from planning the execution of sets of threads based on what interferes with each other to allocate the resources of the system, among all the processes in execution, considering fairness and performance.

All these solutions have a common meeting point: the necessity to estimate the deceleration suffered by an application when it competes with others for shared resources; in this case, the access to main memory.

While wasted bandwidth value when a process is in execution simultaneously with others can be measured, the challenge is to make a real-time estimation about the bandwidth that would consume the same application if it would run alone in the same system. The different memory access patterns of the different applications that are running at the same time in a system, the location in memory of the resources requested by these and the existence of main memory access algorithms, existing in the memory controller, make it difficult to estimate [7].

We can’t forget that the use of resources changes over time, as the available resources [13], so it is mandatory to make frequent and effective estimates.

In the context of cloud computing, where several applications are normally run simultaneously, it is imperative to ensure quality of service to all users. The hardware itself does not give applications a fraction of the shared resources proportional to the priority set for each of them. It is possible that processes running simultaneously with the same priority could offer a performance very different from each other [14].

What is proposed in this work is an algorithm for managing shared resources in real time to optimize fairness [10, 14] during the execution of tasks with the same priority. Intel CAT [15] technology allows the software to manage the last level cache used by applications. The algorithm is designed to perform an equal distribution of this resource and to control the use of the main memory bandwidth between execution tasks by techniques related to deactivation of cores.

With the open source monitoring tool PMCTrack [13, 16] and with the available hardware different techniques will be studied [5, 7, 10, 9, 6] to carry out an efficient planning of tasks prioritizing fairness by managing the use of the available bandwidth.

Project goals

In order to make fair management of shared resources, it was necessary to carry out prior research work related to estimate the degradation in performance suffered by running tasks when having to compete for their use.

This work takes models and conclusions obtained by other authors [5] in order to estimate this degradation in terms of the degradation they suffer when having to compete for main memory access. Therefore, it was mandatory to validate each of them as far as possible. It was possible thanks to PMCTrack possibilities; its command line interface offers to user space information related to distinct metrics that can be interesting to resolve any problem.

After learning about different techniques proposed [5, 7, 10, 9, 6] related to the allocation of shared resources was carried out the development of an algorithm for management of shared resources in real time in the Linux kernel. This algorithm uses interfaces provided by PMCTrack [13, 16] to monitor metrics provided by hardware counters and the support of Intel CAT [15] and Intel MBM technologies [17]; last one was used to monitor the bandwidth consumed by running applications.

Benchmarks present at SPEC2000 and SPEC2006 suites [18, 19] allow the possibility of evaluate several techniques studied, and then applied, with diverse workloads.

Once the development of the algorithm in a module of the Linux kernel was finished, the experimental phase was carried out to verify that the techniques used improve fairness. The Het-Harness framework was used at this stage; it allows the systematically launch of on demand workloads.

Work plan

The following steps have been taken to reach these goals:

1. Utilization of PMCTrack's [13, 16] command line interface.
2. Learn about Het-Harness framework to carry out the experimental phases.
3. Learning Python as a scripting language for data processing.
4. Study of different techniques proposed by other authors [5, 7, 10, 9, 6] in order to design algorithms. Elaboration, processing and interpretation of associated experiments.
5. Training on Linux kernel programming skills.
6. Utilization of Weka [20] software for machine learning purposes.
7. Development of shared resources management algorithm in a Linux kernel module.
8. Evaluation of the results offered by the development.

Exposed work plan is orientative. Along time it has been frequent to know more details about the PMCTrack [13, 16] tool. In the same way, before a previous step to learn about them, C and Python [21, 22, 23, 24] new resources were used. The search for alternative techniques has also been frequent when, validated first and then employed have not been fully satisfactory to achieve the goal of this work.

Structure of the document

Following chapters maintain the established planning of this work. The division of the contents is based on the different phases that this work has required. In each one of them, the motivation of a particular task is explained, the steps taken to carry out it and the conclusions that are relevant.

Finally, it is a work that has required multiple, mostly small, developments. Then, the most relevant sections of these will also be presented. Exceptional case is a chapter dedicated exclusively to the design and implementation of the shared resources management real time algorithm due to its complexity, extension and transcendence in this project.

Bellow, a briefly description of the content of each chapter:

Chapter 2 exposes a research about the accuracy of the proposed models [5] that allow to estimate the degradation suffered by applications when they have to compete for the shared resources of the system where they are running. In addition, recurring concepts throughout this document are defined.
Chapter 3 presents the experimental platform, the hardware extensions Intel CAT [15] and Intel MBM [17] and the support provided by PMCTrack [13, 16] to carry out this work.
Chapter 4 presents the design and the development of the shared resources management real time algorithm with high level of detail.
Chapter 5 shows the results obtained using the proposed algorithm.

Appendix B - Conclusions and future work

Conclusions

During the development of this work it has been possible to observe the behaviour of different applications depending on the available resources. In cloud computing, provide quality of service to all users of a platform can be a challenge. The servers used for this purpose run applications that consume shared resources in very different ways.

This work has proposed an algorithm for managing shared resources in real time focusing on processes that have the same priority running simultaneously in the same system. The algorithm tries to manage shared resources utilization in order to optimizing fairness and providing quality of service.

The availability of an experimental platform with Intel CAT [15] and Intel MBM [17] technologies, the objective has been to manage the use of shared resources when applications that make intensive use of the last level cache memory and the main memory bandwidth are running simultaneously on the same system. It was possible because we could estimate the degradation suffered by this type of tasks [morad2016efs] by measuring their bandwidth usage.

The open source monitoring tool PMCTrack [13, 16] allows users to easily monitor these events from a command line interface. This was used in the early stages of this work in order to detect trends after applying restrictions to processes about shared resources. Later, the support of this tool was used for the development of a Linux kernel module where the shared resources management real time algorithm was embedded. This tool offers to the programmer an interface to consider events obtained from the hardware.

In order to evaluate the accuracy of the proposed algorithm, a framework was used, Het-Harness. This framework allowed to launch several workloads in a systematic way in the experimental platform. The results of the execution of each one of them were obtained. SPEC CPU 2000 and SPEC CPU 2006 suites were used to evaluate the implementation. Relevant conclusions about the implementation were obtained.

Latest results obtained from real time management of the use of shared resources confirm significant trends of equalizing the degradation suffered by tasks with the same priority.

Future work

The conclusions presented are related to exploit the potential of the tools used, both hardware and software, due to different observations presented at the experimentation phase.

During the development of this project it has been possible to observe the behavior of the applications according to the resources provided to them. Using performance metrics and Intel MBM [17] technology support it has been possible to establish a profile for each running process in order to prioritize the use of shared resources based on it. We must not forget that the tasks, during their execution, experiments different phases so it is necessary to make a continuous monitoring for an efficient manage of the shared resources. Using these metrics, it has been possible to establish a profile that relates those applications that stop consuming memory bandwidth when they have enough space at the last cache level as opposed to those that made use the same amount of memory bandwidth regardless of the availability of the available cache.

- Continue using Intel CAT [15] technology in order to prior the use of the last cache level to those applications where better use of this resource can be detected. This will increase in some cases the performance of certain processes while those that make intensive use of the main memory bandwidth will also be improved thanks to low contention situations.

Thanks to Intel MBM [17] and hardware counters it has been possible to estimate the degradation suffered by applications that make intensive use of the main memory bandwidth to apply restrictions on the use of this resource. The hardware used does not limit the use of available bandwidth by itself and this task was delegated to the software.

- Hardware capable for apply these restrictions by itself when detects that the applications reach the established limit may favor the algorithm proposed in this work. It would be very interesting to evaluate it in this context.

When a process does not consume a significant amount of main memory bandwidth, the application of the proposed algorithm does not transcendental. In this context it can not estimate the degradation in performance suffered by that kind of applications.

- The study of techniques about to monitor metrics that could be relevant for better estimations in the context presented and that can also be implemented in the Linux kernel could be useful.

Distinguishing these different profiles of applications presented can lead to an efficient management of shared resources favoring the quality of service that a platform can provide.

Bibliografía

[1] J. L. Hennessy y D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[2] P. Machanick, «Approaches to addressing the memory wall», *School of IT and Electrical Engineering, University of Queensland*, 2002.

[3] F. Guo, Y. Solihin, L. Zhao, y R. Iyer, «Quality of service shared cache management in chip multiprocessor architecture», *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 7, n.º 3, p. 14, 2010.

[4] A. Jaleel, H. H. Najaf-Abadi, S. Subramaniam, S. C. Steely, y J. Emer, «CRUISE: cache replacement and utility-aware scheduling», en *ACM SIGARCH Computer Architecture News*, 2012, vol. 40, pp. 249-260.

[5] 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.

[6] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, y O. Mutlu, «MISE: Providing performance predictability and improving fairness in shared main memory systems», en *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, 2013, pp. 639-650.

[7] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, y L. Sha, «Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms», en *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, 2013, pp. 55-64.

[8] D. Xu, C. Wu, y P.-C. Yew, «On mitigating memory bandwidth contention through bandwidth-aware scheduling», en *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, 2010, pp. 237-248.

[9] D. R. Hower, H. W. Cain, y C. A. Waldspurger, «PABST: Proportionally Allocated Bandwidth at the Source and Target», en *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, 2017, pp. 505-516.

[10] 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

ACM Sigplan Notices, 2010, vol. 45, pp. 335-346.

[11] T. Marinakis, A.-H. Haritatos, K. Nikas, G. Goumas, y I. Anagnostopoulos, «An efficient and fair scheduling policy for multiprocessor platforms», en *Circuits and Systems (ISCAS), 2017 IEEE International Symposium on*, 2017, pp. 1-4.

[12] D. Xu, C. Wu, P.-C. Yew, J. Li, y Z. Wang, «Providing fairness on shared-memory multiprocessors via process scheduling», en *ACM SIGMETRICS Performance Evaluation Review*, 2012, vol. 40, pp. 295-306.

[13] J. C. Saez, A. Pousa, R. Rodriíguez-Rodriíguez, 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.

[14] A. G. García, «Optimización de justicia y rendimiento en procesadores multicore asimétricos mediante planificación consciente de la contención», 2018.

[15] *Intel Cache Allocation Technology*. «https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology».

[16] *PMCTrack's Official Website*. «https://pmctrack.dacya.ucm.es».

[17] *Intel Memory Bandwidth Monitoring*. «https://software.intel.com/en-us/articles/introduction-to-memory-bandwidth-monitoring».

[18] J. L. Henning, «SPEC CPU2000: Measuring CPU performance in the new millennium», *Computer*, vol. 33, n.º 7, pp. 28-35, 2000.

[19] J. L. Henning, «SPEC CPU2006 benchmark descriptions», *ACM SIGARCH Computer Architecture News*, vol. 34, n.º 4, pp. 1-17, 2006.

[20] *Weka 3: Data Mining Software in Java*. «https://www.cs.waikato.ac.nz/ml/index.html».

[21] *Elixir Cross References*. «https://elixir.bootlin.com/linux/latest/source/kernel».

[22] R. Love y others, *Linux kernel development*, vol. 2. Novell Press Indianapolis, IN, USA, 2005.

[23] *ftrace - Function Tracer*. «https://www.kernel.org/doc/Documentation/trace/ftrace.txt».

[24] *Scientific Computing Tools for Python*. «https://www.scipy.org».

[25] *Intel Xeon Processor specifications*. «https://ark.intel.com/products/92986/Intel-Xeon-Processor-E5-2620-v4-20M-Cache-2_10-GHz».

[26] H. Yun, R. Mancuso, Z.-P. Wu, y R. Pellizzoni, «PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms», en *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, 2014, pp. 155-166.

[27] I. Corparation, «Intel (R) 64 and IA-32 Architectures Software Developer's Manual», *Combined Volumes, Dec*, 2016.

[28] *Using Intel® VTune™ Amplifier XE to Tune Software on the 5th generation Intel® Core™ process*. «https://software.intel.com/sites/default/files/managed/65/c1/Using_Intel_

VTune_Amplifier_XE_on_5th_Generation_Intel_Core_Processors_1.0.pdf».

[29] J. C. Saez, A. Pousa, F. Castro, D. Chaver, y M. Prieto-Matias, «Towards completely fair scheduling on asymmetric single-ISA multicore processors», *Journal of Parallel and Distributed Computing*, vol. 102, pp. 115-131, 2017.

[30] *Intel Broadwell microarchitecture hardware events*. «http://download.01.org/perfmon/BDX/broadwellx_core_v13.json».

[31] J. Feliu, J. Sahuquillo, S. Petit, y J. Duato, «Perf Fair: a Progress-Aware Scheduler to Enhance Performance and Fairness in SMT Multicores», *IEEE Transactions on Computers*, vol. PP, n.º 99, pp. 1-1, 2016.

[32] J. C. Saez, A. Pousa, F. Castro, D. Chaver, y M. Prieto-Matias, «Exploring the Throughput-Fairness Trade-off on Asymmetric Multicore Systems», en *Proceedings of Euro-Par 14: Parallel Processing Workshops*, 2014, pp. 326-337.