Análisis del potencial del particionado de cache en sistemas multinúcleo para garantizar aislamiento entre aplicaciones y calidad de servicio en la nube

Analyzing the potential of cache partitioning in multicore systems for application isolation and quality of service in the cloud

Trabajo de Fin de Grado
Curso 2023–2024

Autores
Diego Pellicer Lafuente
Yikang Chen

Directores
Juan Carlos Saez Alcaide
Carlos Bilbao

Grado en Ingeniería de Computadores
Facultad de Informática
Universidad Complutense de Madrid

Análisis del potencial del particionado de cache en sistemas multinúcleo para garantizar aislamiento entre aplicaciones y calidad de servicio en la nube
Analyzing the potential of cache partitioning in multicore systems for application isolation and quality of service in the cloud

Trabajo de Fin de Grado en Ingeniería de Computadores

Autores
Diego Pellicer Lafuente
Yikang Chen

Directores
Juan Carlos Saez Alcaide
Carlos Bilbao

Convocatoria: *Junio 2024*

Grado en Ingeniería de Computadores
Facultad de Informática
Universidad Complutense de Madrid

21 de Mayo de 2024

Dedicatoria

Yo, Diego Pellicer, quiero agradecer en especial a mis padres y hermanos por apoyarme en cada momento. Sin ellos no sería posible llegar hasta quí y no puedo estar más que agradecido con ellos. También quiero agradecer a mi grupo de amigos LPC por estar siempre ahí conmigo y poder contar siempre con ellos. También dedicar este TFG a todos los grandes amigos que he hecho a lo largo de la carrera, que han hecho de esta un recuerdo muy especial. Por último este TFG va dedicado a mis abuelos, los cuales han estado siempre ahí para mí y no me puede hacer más feliz que puedan verme lograr terminar esta carrera.

Yo, Yikang Chen, dedico este trabajo a todas las personas de la Facultad de Informática que nos han acompañado. Además, quiero expresar mi agradecimiento a mis padres por todo el apoyo que me han brindado a lo largo de mi carrera.

Agradecimientos

Queremos expresar nuestro agradecimiento a los directores del TFG, quienes nos han brindado una invaluable ayuda a lo largo de todo el proyecto, resolviendo nuestras dudas y orientándonos en cada etapa. En especial, queremos agradecer a Juan Carlos por su dedicación y compromiso con nuestro trabajo de fin de grado. Su pasión por la investigación, tanto como profesor como director, ha sido una fuente de inspiración para nosotros.

También queremos agradecer a nuestras familias por su apoyo, sin ellas no ha sido posible llegar hasta aquí. Gracias por brindar vuestro apoyo tanto dentro como fuera de la carrera.

Resumen

Análisis del potencial del particionado de cache en sistemas multinúcleo para garantizar aislamiento entre aplicaciones y calidad de servicio en la nube

Los procesadores multinúcleo (CMPs) constituyen actualmente la arquitectura dominante en sistemas de propósito general, y están presentes en un amplio especto de productos comerciales, desde dispositivos móviles a servidores en centros de datos. Maximizar la utilización del número creciente de núcleos de estas arquitecturas es crucial para incrementar los beneficios económicos, especialmente en centros de datos en la nube. Para abordar este desafío, los proveedores de *cloud* ejecutan múltiples servicios y aplicaciones de uno o varios clientes en el mismo servidor físico. Esto se combina con el lanzamiento cargas de trabajo propias de los proveedores de cloud durante períodos de baja demanda para maximizar la utilización de recursos.

Lamentablemente, la ejecución simultánea de diversas aplicaciones en un sistema CMP a menudo provoca una degradación de rendimiento desigual y difícil de predecir en las aplicaciones debido a la contención de recursos compartidos. Esta contención surge porque los núcleos de los sistemas CMP típicamente comparten recursos críticos como la caché de último nivel o el ancho de banda de memoria. No arbitrar explícitamente la competición por estos recursos entre aplicaciones provoca la degradación sistemática del rendimiento y de la justicia en el sistema, y puede afectar de forma muy negativa a parámetros críticos del funcionamiento de los servicios en la nube, como la latencia observada. Para hacer frente a estos problemas, los principales fabricantes de procesadores han optado en los últimos años por la inclusión en sus productos comerciales de una serie de extensiones hardware como, por ejemplo, el soporte para particionado de la caché de último nivel.

El objetivo principal de este trabajo es realizar evaluaciones experimentales para analizar el potencial del particionado de caché como mecanismo de aislamiento entre aplicaciones en un servidor multinúcleo. Las conclusiones de este análisis buscan sentar las bases para la construcción de un gestor de recursos implementado dentro del kernel del sistema operativo, particularmente en el kernel Linux. Para llevar a cabo nuestro análisis ha sido necesario realizar extensiones en una serie de herramientas, y con ello posibilitar el lanzamiento y monitorización de cargas de trabajo formadas por aplicaciones en contenedores.

Palabras clave

Contención de recursos, Multicore, Particionado de cache, cloud, HPC, Linux kernel, PMCSched, CloudSuite, Docker.

Abstract

Analyzing the potential of cache partitioning in multicore systems for application isolation and quality of service in the cloud

Multi-core processors (CMPs) are currently the dominant architecture in general-purpose systems, and are present in a wide range of commercial products, from mobile devices to servers in data centers. Maximizing the utilization of the growing number of cores in these architectures is crucial to increase economic benefits, especially in cloud data centers. To address this challenge, cloud providers run multiple services and applications from one or more clients on the same physical server. This is combined with launching cloud providers' own workloads during periods of low demand to maximize resource utilization.

Unfortunately, concurrent execution of multiple applications on a CMP system often results in uneven and hard-to-predict application performance degradation due to shared resource contention. This contention arises because the cores of CMP systems typically share critical resources such as last-level cache or memory bandwidth. Failure to explicitly arbitrate competition for these resources between applications leads to systematic performance and fairness degradation in the system, and can strongly negatively affect critical cloud service performance metrics such as observed latency. To address these problems, major processor manufacturers have in recent years opted for the inclusion in their commercial products of a number of hardware extensions, such as, for example, support for last-level cache partitioning.

The main objective of this work is to perform experimental evaluations to analyze the potential of cache partitioning as an isolation mechanism between applications in a multicore server. The conclusions of this analysis seek to lay the foundations for the construction of a resource manager implemented within the operating system kernel, particularly in the Linux kernel. In order to carry out our analysis it has been necessary to make extensions to a number of tools, and thus enable the launching and monitoring of workloads consisting of containerized applications.

Keywords

Containment of resources, Multicore, Cache partitioning, cloud, HPC, Linux kernel, PMCSched, CloudSuite, Docker.

Índice

1. Introducción **1**

1.1. Motivación 1

1.2. Objetivos 3

1.3. Plan de trabajo 4

1.4. Estructura de la memoria 5

2. Herramientas Utilizadas **7**

2.1. PMCSched 7

2.1.1. Framework PMCSched 8

2.2. Het-Harness 14

2.2.1. Arquitectura de Het-Harness 15

2.3. Cloudsuite 18

2.3.1. Adaptaciones de Cloudsuite 20

3. Adaptaciones en Het-Harness **23**

3.1. Soporte inicial de contenedores 24

3.2. Soporte para aplicaciones Cloudsuite 26

3.3. Sistema de scripting 27

3.3.1. Ejemplo de un *benchmark script*, de Cloudsuite 27

4. Adaptaciones de PMCSched **31**

4.1. Contenedores como aplicación multihilo 31

5. Análisis Experimental **35**

5.1. Plataforma Experimental 35

5.1.1. Plataforma Broadwell 35

5.1.2. Plataforma Skylake 36

5.2. Benchmarks 37

5.2.1. HPC 37

5.2.2. CloudSuite 38

5.3. Clases de *benchmarks* 39

5.4. Fase experimental 40

5.4.1. Barridos de R/TPS para *benchmarks* de CloudSuite 41

5.4.2. Experimentos de caracterización I: Latencia/EFFTC/BW 44

5.4.3. Experimentos de caracterización II 50

5.4.4. Experimentos con particionado de caché estático 59

6. Extensión del sistema operativo para clasificación de aplicaciones **67**

6.1. Análisis experimental y heurística 67

6.2. Plugin de caracterización de aplicaciones 81

7. Conclusiones y Trabajo Futuro **91**

Introduction **95**

Conclusions and Future Work **101**

Contribuciones Personales **103**

Bibliografía **107**

Índice de figuras

1.1. Diagrama de Gantt 5

2.1. Esquema estructura de PMCTrack y PMCSched 8

2.2. Arquitectura de Het-Harness 16

2.3. Diagrama de dependencias de los *benchmarks* de Cloudsuite 19

2.4. *Dockerfile* adaptado del servidor de Data serving relational 21

3.1. Flujo de la carga de trabajo multi-aplicación antes de la actualización 23

3.2. Flujo de la carga de trabajo multi-contenedor con aplicaciones convencionales 25

3.3. Flujo de la carga de trabajo multi-contenedor con aplicaciones de Cloudsuite 26

3.4. Las variables del inicio del script del *benchmark* de ejemplo 28

3.5. La opción de *start* del script del *benchmark* de ejemplo 29

3.6. La opción de *exec* del script del *benchmark* de ejemplo 30

3.7. La opción de *stop* del script del *benchmark* de ejemplo 30

5.1. Topología del Intel(R) Xeon(R) CPU E5-2620 v4 36

5.2. Topología del Intel(R) Xeon(R) Gold 6138 37

5.3. Esquema distribución de los benchmarks 41

5.4. Gráficas de latencia de Data Serving Relational según el TPS 42

5.5. Gráficas de latencia de Data Serving MySQL según el TPS 42

5.6. Gráfica de latencia de Data Caching según el TPS 43
5.7. Gráficas de BW/EFFTC de BLAST 44
5.8. Gráficas de BW/EFFTC de BT 45
5.9. Gráficas de BW/EFFTC de PBBCache 45
5.10. Gráficas de BW/EFFTC de RNASEQ 46
5.11. Gráficas de BW/EFFTC de LU 46
5.12. Gráficas de BW/EFFTC de FFTW3D 47
5.13. Gráficas de BW/EFFTC de bodytrack 47
5.14. Gráficas de BW/EFFTC de Graph Analytics modo PR 48
5.15. Gráficas de BW/EFFTC de Graph Analytics modo TC 48
5.16. Gráficas de BW/EFFTC de Data Serving Relational modo OLTP-RW 49
5.17. Gráficas de Latencia de Data Serving Relational modo OLTP-RW . . 49
5.18. Gráficas de BW y Slowdown de BT 51
5.19. Gráficas de BW y Slowdown de BLAST 51
5.20. Gráficas de BW y Slowdown de bodytrack 52
5.21. Gráficas de BW y Slowdown de PBBCache 52
5.22. Gráficas de BW y Slowdown de RNASeq 53
5.23. Gráficas de BW y Slowdown de LU 53
5.24. Gráficas de BW y Slowdown de FFTW3D 54
5.25. Gráficas de BW y Slowdown de Graph Analytics (modo PR) 55
5.26. Gráficas de BW y Slowdown de Graph Analytics (modo CC) 55
5.27. Gráficas de BW y Slowdown de Graph Analytics (modo TC) 56
5.28. Gráficas de BW y Slowdown de Data Serving Relational (modo OLTP-RW) 56
5.29. Gráfica de latencia de Data Serving Relational (modo OLTP-RW) . . 57
5.30. Gráficas de BW y Slowdown de Data Serving Relational (TPC-C) . . 57
5.31. Gráfica de latencia de Data Serving Relational (TPC-C) 58
5.32. bodytrack vs postgresql-oltp 60

5.33. LU vs postgresql-oltp 61

5.34. BT vs postgresql-oltp 61

5.35. Comparación de latencia del *benchmark* postgresql-oltp en los exp 1, 2 y 3 62

5.36. bodytrack vs graph_analytics-tc 63

5.37. LU vs graph_analytics-tc 63

5.38. BT vs graph_analytics-tc 64

5.39. bodytrack vs graph_analytics-pr 65

5.40. LU vs graph_analytics-pr 65

5.41. BT vs graph_analytics-pr 65

6.1. Matriz de correlación para las métricas 70

6.2. Matriz de confusión de la predicción del árbol de decisión 71

6.3. Árbol de decisión de altura 4 72

6.4. Distribución espacial de las distintas clases 73

6.5. Resultados k-means 74

6.6. Medias de las métricas según su clase 75

6.7. Máximos y mínimos l2reuse según vías de cache 76

6.8. Máximos y mínimos stalls_l2_miss según número de vías 77

6.9. Máximos light-sharing frente a mínimos aplicaciones cache-sensitves y streamings llcmpki 78

6.10. Máximos light-sharing frente a mínimos cache-sensitves y streamings llcmpkc 79

6.11. Media de la diferencia entre llcrpkc y llcmpkc 80

6.12. Gráficas de la clasificación mediante métricas 1 87

6.13. Gráficas de la clasificación mediante métricas 2 88

7.1. Gantt Chart 99

Índice de tablas

5.1. Aplicaciones con las medidas escogidas y su latencia 43

5.2. Tabla de experimentos con mezcla de aplicaciones Cloud y HPC . . . 59

6.1. PMCs usados para el calculo de métricas 68

6.2. Métricas usadas 68

Capítulo 1

Introducción

"La calidad en un servicio o producto no es lo que pones en él. Es lo que el cliente obtiene de él."
— Peter Drucker

1.1. Motivación

La arquitecturas multinúcleo o CMPs (*Chip Multi-Processors*) son hoy en día la opción de diseño predominante para los sistemas de propósito general en múltiples sectores de la industria. En particular, los CMPs constituyen la arquitectura de facto para servidores en centros de datos en la nube [32, 17, 23]. También son un componente esencial de las plataformas HPC (*High-Performance Computing*) [33]. Gracias a los últimos avances en tecnología y diseño de procesadores, muchos sistemas CMP integran ya cientos de núcleos. Por ejemplo, los servidores basados en el procesador AMD EPYC 9654[1], cuentan con 192 núcleos en configuraciones de doble *socket*.

Aprovechar al máximo los núcleos disponibles y, más generalmente, maximizar la utilización de recursos en los centros de datos en la nube es crítico para reducir los costes de los proveedores de *cloud*, y por tanto resulta clave para aumentar sus ingresos [9]. Lograr esto constituye un reto significativo, y más teniendo en cuenta que las aplicaciones típicas de la nube suelen tener una carga de trabajo muy variable a lo largo del día (p.ej., debido a las fluctuaciones en el número de clientes que hacen uso de un servicio/portal web a lo largo del tiempo). Nótese que la mayoría de estas aplicaciones son además servicios críticos en latencia (LC: *Latency Critical*), como los motores de búsqueda o muchos de los servicios basados en inteligencia artificial. En el ámbito de HPC, existen muchas aplicaciones que tampoco utilizan plenamente los núcleos disponibles en un sistema CMP, debido al desequilibrio de carga entre los procesos/hilos de la aplicación, o a la presencia de otros cuellos de botella de

[1] `https://www.amd.com/es/products/cpu/amd-epyc-9654`

escalabilidad [14, 33].

Para maximizar la utilización de los sistemas CMPs, los proveedores de *cloud* típicamente ejecutan múltiples servicios y aplicaciones de uno o varios clientes en el mismo servidor físico – práctica conocida como *multi-tenancy*. Además, para incrementar aún más el grado de uso de los recursos durante los períodos de baja utilización de los servidores, los propios proveedores suelen lanzar cargas de trabajo propias en el mismo conjunto de servidores dedicados a ejecutar las cargas de trabajo de los clientes [23, 27, 16]. Al gestionar dinámicamente los recursos de los distintos servidores, estas cargas de trabajo de los proveedores se consideran no críticas (BE: *Best Effort*), y por tanto su ejecución puede aplazarse cuando resulta estrictamente necesario [34]. En este Trabajo Fin de Grado (TFG), se considera, por el contrario, la utilización de cargas de trabajo HPC como vía para incrementar el grado de utilización de los servidores en la nube en situaciones de demanda baja o moderada. El paralelismo inherente presente en muchas de estas aplicaciones, hace que resulten especialmente adecuadas para hacer uso efectivo de múltiples núcleos inactivos que pueda haber en el sistema [29].

Cabe destacar que ejecutar distintas aplicaciones de forma simultánea o concurrente en un sistema multinúcleo suele provocar una degradación del rendimiento desigual y difícil de predecir entre aplicaciones, debido al fenómeno de *la contención de recursos compartidos* [35, 24]. Esta clase de contención surge por el hecho de que los núcleos en un sistema CMP actual comparten recursos críticos con otros núcleos vecinos, como una caché de último nivel (LLC: *Last-Level Cache*), canales de memoria o controladores DRAM. No tomar ninguna medida para mitigar los efectos negativos derivados de la competencia por estos recursos entre aplicaciones suele ocasionar la degradación sistemática del rendimiento y provoca injusticia en el uso del sistema (*unfairness*) [13, 25, 28]. Adicionalmente, esto puede suponer el incumplimiento de los acuerdos de nivel de servicio (SLAs: *Service Level Agreements*) asociados a distintas aplicaciones y servicios en la nube [23, 32]. Por ejemplo, se ha demostrado que la latencia efectiva de un servicio *cloud* puede llegar a superar sustancialmente su valor admisible en situaciones severas de contención de recursos [17, 23].

Para permitir que el software de sistema pueda mitigar de forma efectiva los efectos nocivos de la contención, los principales fabricantes de procesadores han optado en los últimos años por la inclusión de una serie de extensiones hardware para gestión dinámica de los recursos compartidos en sus productos comerciales. En el caso de Intel y AMD, estas extensiones se conocen comercialmente como *Intel Resource Director Technology*[2] y *AMD64 Platform QoS Extensions*[3]. Ambas colecciones de extensiones permiten, entre otras cosas, imponer cuotas por aplicación en el uso de recursos compartidos mediante el particionado de la caché de último nivel o la limitación en el consumo del ancho de banda de memoria [26, 30].

[2]https://cdrdv2-public.intel.com/789566/356688-intel-rdt-arch-spec.pdf
[3]https://developer.amd.com/wp-content/resources/56375.pdf

1.2. Objetivos

El principal objetivo de este TFG es llevar a cabo una evaluación experimental para analizar el potencial del particionado de caché para ofrecer aislamiento entre aplicaciones en un servidor multinúcleo. Este análisis se ha realizado sobre un entorno donde las distintas aplicaciones/servicios se ejecutan en contenedores independientes sobre GNU/Linux, ya que este es un escenario habitual de despliegue de aplicaciones en la nube. Cabe destacar que el análisis de este proyecto pretende sentar las bases para la construcción de un gestor de recursos compartidos implementado en el sistema operativo, y más específicamente en el kernel Linux. Por este motivo, gran parte de los experimentos realizados se basan en la utilización de extensiones del sistema operativo, algunas de ellas implementadas como parte de este trabajo.

Para poder alcanzar el objetivo global del proyecto, ha sido necesario perseguir los siguientes objetivos específicos:

- **Desarrollo de una infraestructura para el lanzamiento y monitorización de aplicaciones en contenedores:** En este proyecto se ha optado por utilizar la suite Cloudsuite [1, 19], que consta de un conjunto de *benchmarks* representativos de aplicaciones en la nube, como cargas de bases de datos en disco o en memoria, así como de procesamiento de datos masivos. En esta suite, cada *benchmark* a ejecutar requiere el despliegue previo de uno o varios contenedores, con componentes que trabajan de forma cooperativa, y exigen una cuidadosa preparación de datos (*warmup*) así como el arranque de distintos servicios clave para su funcionamiento.

  La realización de experimentos automatizados con uno o varios de estos *benchmarks* (algunos multi contenedor), ha requerido el desarrollo de una infraestructura específica para posibilitar el lanzamiento y monitorización de cargas de trabajo incluyendo estas aplicaciones. Para ello, en el proyecto se ha usado como base la herramienta *Het-Harness*, utilizada internamente en el grupo de investigación ArTeCS de la UCM. La funcionalidad de esta herramienta se ha extendido de forma sustancial para posibilitar nuestro análisis con aplicaciones *containerizadas*, tanto HPC como *cloud*.

- **Realizar la caracterización de un conjunto representativo de aplicaciones *cloud* y HPC:** Antes de proceder a analizar el impacto del particionado de caché, fue necesario llevar a cabo una caracterización del comportamiento de un conjunto representativo de aplicaciones *cloud* y HPC. En particular, se ha prestado especial atención a monitorizar el comportamiento de estos *benchmarks* en lo que respecta al uso de los recursos compartidos entre núcleos (caché de último nivel y ancho de banda de memoria), así como a su grado de utilización de la CPU a lo largo del tiempo.

  Para extraer este tipo información –parte de la cual está accesible de forma directa solo desde dentro del sistema operativo– se ha hecho uso del framework

open source PMCSched [14]. Aunque este framework ya proporciona soporte para monitorización del uso de la LLC y del ancho de banda, fue necesario extender su funcionalidad para manejar como un todo los procesos de un mismo contenedor, lo cual es un requisito imprescindible para llevar a cabo nuestro análisis.

- **Análisis de la contención de recursos al usar mezclas de aplicaciones Cloud y HPC:** Fruto de los resultados obtenidos en el objetivo anterior, se obtiene una clasificación de aplicaciones en distintas categorías. Esta clasificación da paso a la primera fase de nuestro análisis del impacto del particionado de cache. Para esta fase, se plantea la construcción de un conjunto de mezclas formadas por cargas cloud y HPC, y se analiza la degradación en rendimiento y/o latencia de las distintas aplicaciones o servicios. Asimismo, se estudia el impacto de particionar la LLC de forma estática para mitigar el efecto de la contención de recursos compartidos.

- **Construcción de un mecanismo de clasificación dinámica de aplicaciones basado en contadores hardware:** Como fase final de nuestro análisis, se plantea la captura de distintas métricas de rendimiento usando contadores hardware de monitorización del rendimiento del procesador [31], para llevar a cabo una clasificación de aplicaciones en tiempo de ejecución. Con esta clasificación dinámica, y la implementación de extensiones en el sistema operativo para ponerla en práctica, se pretende sentar las bases para desarrollar estrategias automáticas de particionado de la LLC desde el propio sistema operativo (funcionalidad que actualmente no ofrecen los sistemas operativos de propósito general).

1.3. Plan de trabajo

Para alcanzar los objetivos específicos del trabajo se realizaron las tareas descritas a continuación, cuya planificación temporal se resume en el diagrama de la figura 7.1:

T1 Lectura de la documentación de Cloudsuite e instalación de la suite en los distintos servidores utilizados.

T2 Familiarizarse con el framework PMCsched, y proceder a su despliegue en las distintas plataformas multinúcleo utilizadas en el proyecto.

T3 Completar distintos tutoriales de introducción a la herramienta Het-Harness, y análisis de la documentación propietaria existente sobre esta herramienta.

T4 Desarrollo de las adaptaciones de Cloudsuite para cubrir las necesidades de nuestro proyecto.

T5 Actualización de las herramientas Het-Harness y PMCSched de forma que se adapte a las necesidades de nuestro proyecto.

T6 Análisis de métricas para la caracterización de aplicaciones. Creación de heurísticas para la caracterización de las aplicaciones.

T7 Implementación del método de clasificación en PMCSched.

T8 Realización del experimento de caracterización 1: Ejecución de las aplicaciones de Cloudsuite. Creación de gráficas y análisis de las gráficas creadas.

T9 Realización del experimento de caracterización 2: Ejecutar todas las aplicaciones, tanto *cloud* como HPC, limitándoles su recurso compartido (caché de último nivel). Creación de gráficas y análisis.

T10 Realización del experimento de caracterización 3: Mezcla de aplicaciones *cloud* y HPC. Creación de gráficas y análisis de las gráficas creadas.

T11 Redacción de la memoria.

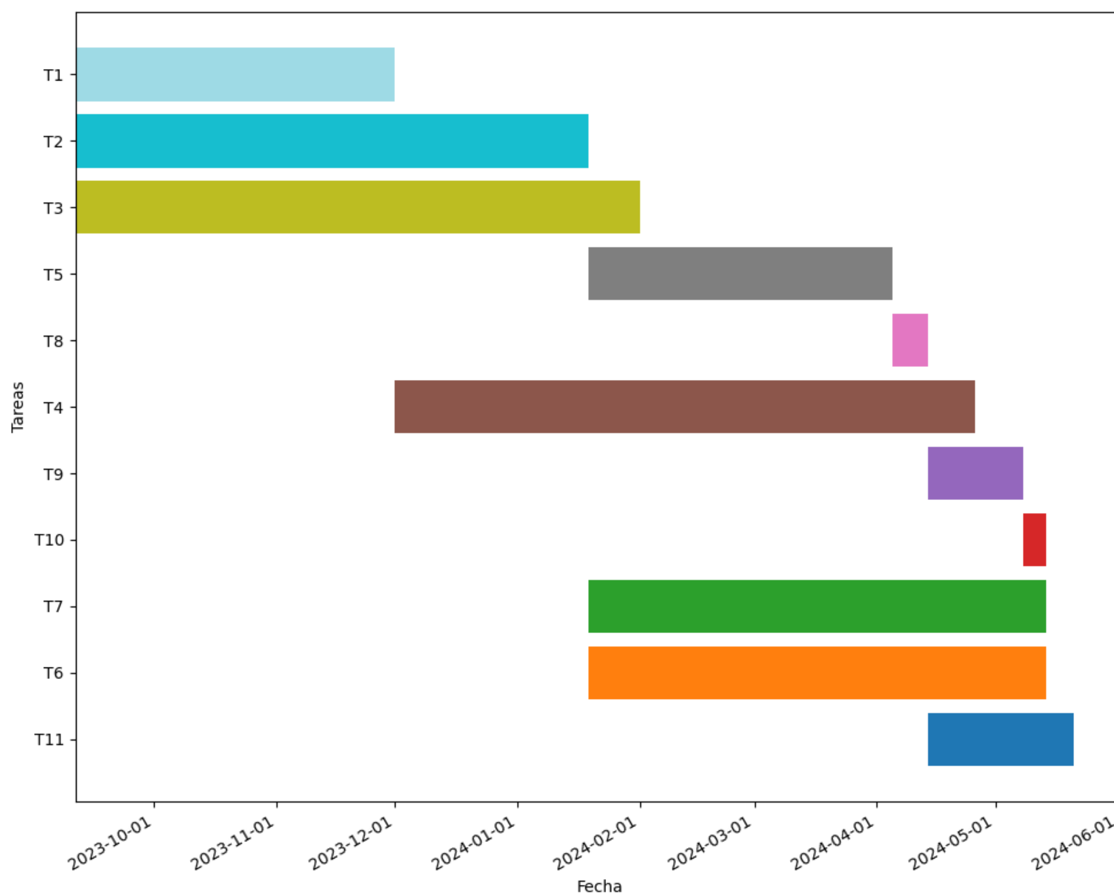


Figura 1.1: Diagrama de Gantt

1.4. Estructura de la memoria

El resto de este documento se estructura de la siguiente forma:

- El capítulo 2 describe las distintas herramientas utilizadas durante la realización de este trabajo. Esta descripción de las herramientas incluye las capas que las componen y su forma de uso.
- El capítulo **??** detalla los cambios realizados en la herramientas Het-Harness para hacer posible el uso de esta con contenedores y aplicaciones cloud.
- El capítulo 4 describe las modificaciones realizadas en el framework PMCSched para un mejor soporte de este en el uso de contenedores.
- El capítulo 5 contiene una descripción de las plataformas experimentales utilizadas así como los distintos experimentos realizados y como se desarrollaron. Para cada experimento realizado se analiza los resultados además de las conclusiones alcanzadas tras este.
- El capítulo 6 detalla los pasos seguidos para desarrollar una heurística para la clasificación de aplicaciones y la creación de una extensión del sistema operativo que incorpora dicha heurística para la clasificación de aplicaciones en tiempo de ejecución. Además este contiene el análisis de los resultados tras la clasificación de *benchmarks* de aplicaciones cloud haciendo uso de este *plugin*.
- El capítulo 7 enumera las conclusiones finales a las que se han llegado a lo largo de este trabajo de fin de grado. Además este capítulo también recoge las posibles líneas de trabajo futuro.

La memoria incluye también tres apéndices. Los apéndices A y B recogen la traducción al inglés de los capítulos de introducción y conclusiones, respectivamente. El apéndice C resume las contribuciones específicas de cada integrante del equipo de este proyecto al TFG realizado.

Capítulo 2

Herramientas Utilizadas

En este capítulo, se procede a explicar las distintas herramientas usadas en el desarrollo de este proyecto. Se pretende con el poner en contexto su estructura, modo de uso así como la importancia de su uso para la realización de este trabajo.

2.1. PMCSched

PMCSched es un framework *open-source* orientado a sistemas operativos para el prototipado de algoritmos de planificación de procesos [14].

Se trata de un framework para el kernel Linux, el cual permite el desarrollo de *plugins*. Para planificación de procesos y gestión de recursos con soporte a nivel de sistema operativo tanto para sistemas multi-core simétricos y asimétricos [1]. Lo que hace interesante el uso de este framework es que no requiere parchear el Kernel Linux para su funcionamiento. Este nos permite incorporar nuevos *plugins* para gestión y planificación de recursos a través de un módulo del kernel que puede ser cargado en kernels sin modificar.

PMCSched es una extensión de la herramienta open source PMCTrack: una trata de una herramienta de monitorización de rendimiento para Linux [31]. A través de PMCTrack podemos obtener información de los contadores de monitorización de rendimiento del procesador o PMCs (Performance Monitoring Counters) desde los *plugins* desarrollados en PMCSched.

[1]Entendemos por sistema multicore asimétrico aquel que incluye distintos tipos de core con el mismo repertorio de intsrucciones (ISA) como pueden ser los procesadores AlderLake y RaptorLake de Intel [22]

2.1.1. Framework PMCSched

El framework PMCSched se implementa como un componente de PMCTrack, esto se logra encapsulando el core de PMCSched como un *módulo de monitorización* de PMCTrack [31].

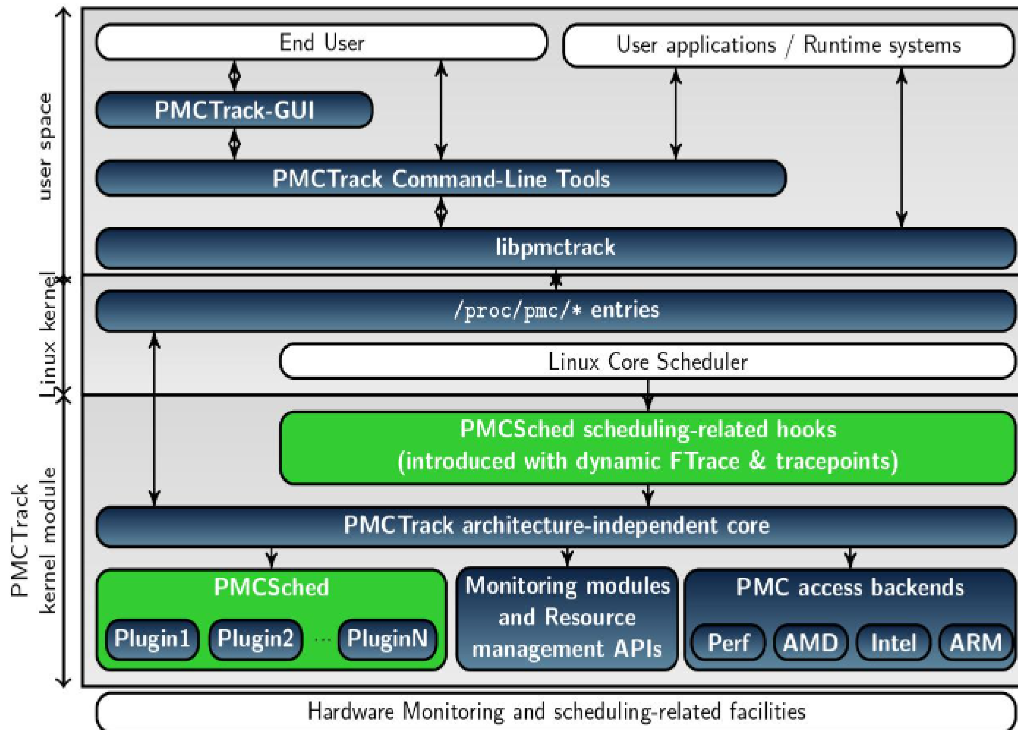


Figura 2.1: Esquema estructura de PMCTrack y PMCSched

La figura 2.1 describe la estructura de PMCTrack y sus distintas capas. En este esquema podemos ver como PMCTrack se trata de un conjunto de componentes que conforma PMCTrack. Algunos de estos componentes se ejecutan en espacio de usuario y otros lo hacen en el kernel. Los componentes de PMCTrack en espacio de usuario, dan al usuario o aplicaciones la posibilidad de poder interactuar con el módulo del kernel de PMCTrack. El módulo del kernel es el que implementa la gran mayoría de la funcionalidad de PMCTrack además de ser el encargado de recoger los distintos valores de los PMCs. También podemos observar que el módulo de PMCTrack a su vez se compone de otros un conjunto de *módulos de monitorización*, PMCSched es un módulo de monitorización más dentro de los componentes del espacio de kernel de PMCTrack. Este módulo de PMCSched a su vez implementa los distintos *plugins* que se desarrollen a través del framework. La última capa de PMCSched que podemos observar en la figura 2.1 es la encargada de que este sea consciente de los eventos de planificación dentro del kernel. En las versiones más recientes este se construye haciendo uso de Ftrace.

Como se ha mencionado anteriormente el framework PMCSched permite el desarrollo de *plugins*. Estos *plugins* implementan una serie de *callbacks* definidas por el framework. Crear un *plugin* es semejante a la creación de un algoritmo de planificación de Linux, a excepción de no tener que modificar el kernel. Cada *plugin* puede implementar distintas funciones, las cuales tienen sus propios atributos y se encargan de manejar eventos específicos.

```
int  (*probe_plugin) (void);
int  (*init_plugin)  (void);
void (*destroy_plugin) (void);
void (*on_exec_thread) (pmon_prof_t* prof);
void (*on_active_thread) (pmcsched_thread_data_t* t);
void (*on_inactive_thread)(pmcsched_thread_data_t* t);
int  (*on_fork_thread) (pmcsched_thread_data_t* t, unsigned char is_new_app);
void (*on_free_thread) (pmcsched_thread_data_t* t, unsigned char is_last_thread);
void (*on_exit_thread) (pmcsched_thread_data_t* t);
void (*sched_kthread_periodic) (sized_list_t* migration_list);
void (*sched_timer_periodic)(void);
int (*on_new_sample) (pmon_prof_t* prof, int cpu, pmc_sample_t* sample, int
    flags,void* data);
int (*on_read_plugin) (char *aux);
int (*on_write_plugin) (char *line);
void (*on_switch_in_thread)(pmon_prof_t* prof, pmcsched_thread_data_t* t,
    unsigned char prof_enabled);
void (*on_switch_out_thread)(pmon_prof_t* prof,pmcsched_thread_data_t* t,
    unsigned char prof_enabled);
void (*on_migrate_thread)(pmcsched_thread_data_t* t, int prev_cpu, int new_cpu);
void (*on_tick_thread)(pmcsched_thread_data_t* t, int cpu);
```

En este conjunto de funciones que todo *plugin* de PMCSched puede implementar, encontramos un struct que se repite en los argumentos de estas. El struct *pmcsched_thread_data_t*, que guarda la información necesaria por hilo para cada hilo registrado por PMCSched, un hilo se encuentra registrado en PMCSched cuando de este se tiene registro de la aplicación a la que pertenece.

Cada una de estas *callbacks* maneja un evento específico el cual nos encargaremos de definir nosotros en nuestro *plugin*. De todas estas *callbacks*, cabe destacar algunas de estas las cuales se van a describir las más relevantes para el propósito de este TFG:

- **on_active_thread:** Esta función se invocará en nuestro *plugin* al momento en el que un hilo pasada de bloqueado o listo para ejecutar.
- **on_inactive_thread:** Esta función maneja el evento en el cual alguno de los hilos que se encontraban activos se vuelve inactivo, deja de estar listo para ejecutar.
- **on_exit_thread:** Se encarga de manejar el momento en el cual un hilo ter-

mina su ejecución.

- **sched_timer_periodic:** A través de un temporizador el cual podemos configurar, se realizan llamadas periódicas a esta función.
- **init_plugin:** Esta función de invoca al momento en el que *plugin* es cargado. Esta se usa para inicializar ciertas estructuras de las cuales haga uso nuestro *plugin*.
- **destroy_plugin:** Se invoca en el momento en que nuestro *plugin* se desactiva. En esta se libera la memoria usada para las estructuras de las cuales hace uso el *plugin*.
- **on_new_sample:** Esta se invoca cada vez que PMCSched obtiene una nueva muestra de valores de PMCs obtenidos vía PMCTrack.

Como podemos observar a través de la *callback* **on_new_sample**, además de poder definir e implementar las distintas *callbacks* que encontramos en el framework. También podemos hacer uso de PMCTrack dentro de nuestro *plugin*, esto nos permite obtener los valores de los PMCs que especifiquemos dentro de nuestro *plugin* e incluso crear métricas a partir de estos. Por ejemplo consideremos el siguiente ejemplo:

```
static const char *pmcstr_config[] =
    {"pmc0,pmc1,pmc2,pmc3=0x2e,umask3=0x4f",NULL};
```

Para poder definir los distintos PMCs tenemos que hacer uso de un formato de bajo nivel definido como formato *raw*. Esto se debe a que el uso de mnemotécnico para definir los distintos contadores hardware a usar, requiere del uso de tablas a partir de las cuales buscar la equivalencia de la métrica a formato *raw*. Esto causa un overhead el cual no se puede permitir en módulos de monitorización del kernel [31]. Para poder definir los valores hexadecimales en *raw* podemos acudir a la documentación facilitada por el fabricante de la CPU de la que vayamos a hacer uso, en caso de Intel puede encontrarse en la siguiente referencia [3].

```
enum event_indexes
{
   INSTR_EVT = 0,
   CYCLES_EVT,
   UNHALTED_REF_CYCLES_EVT,
   LLC_REFERENCES_EVT,
   PMC_EVENT_COUNT
};
```

Además de definir los distintos PMCs que va a implementar nuestro *plugin*, también debemos de definir estos en un enum de eventos. Es importante en este enum asociar a cada contador el evento adecuado para mayor legibilidad y mantenibilidad del *plugin*. Como se puede observar en código, los eventos definidos anteriormente a través del formato *raw*, se tratan de los siguiente eventos:

- **INSTR_EVT:** Se refiere al evento de ejecución de instrucciones.
- **CYCLES_EVT:** Este referencia al evento de ejecución de ciclos.
- **UNHALTED_REF_CYCLES_EVT:** Este evento se refiere a los ciclos de bus.
- **LLC_REFERENCES_EVT:** Este define el evento de referencias a las cache de tercer nivel o LLC.

También podemos definir las métricas que vayamos a crear a través de otro enum.

```
enum metric_indices
{
    IPC_MET = 0,
    LLCRPKI_MET,

    NR_METRICS,
}
```

Al igual que pasaba a la hora de definir los eventos, es importante elegir bien el nombre que daremos a nuestras métricas. Una vez estas están definidas solo queda definir cómo se calculan estas métricas.

```
/* Descriptors for the various performance metrics */
static metric_experiment_set_t metric_description = {
.nr_exps = 1,
.exps = {
    /* Metric set 0 */
    {
        .metrics = {
            PMC_METRIC("IPC", op_rate, INSTR_EVT, CYCLES_EVT,
                1000),
            PMC_METRIC("LLCRPKI", op_rate, LLC_REFERENCES_EVT,
            INSTR_EVT, 1000000),
            },
            .size = NR_METRICS,
          .exp_idx = 0,
    },
}};
```

Como se observa en el código de ejemplo, el valor que tendrán las métricas se definen mediante la macro *PMC_METRIC*. En esta se asocia la métrica que se va a definir como primer argumento y en los siguientes argumentos, se elige la operación a realizar y entre que dos eventos o métricas. En el ejemplo, se usa el operador *op_rate*, el cual se trata de una división entre el primer evento o métrica divido por el segundo; y siendo el dividendo multiplicado por el último argumento que se pasa a la macro.

Las métricas que se han implementado representan los siguientes valores:

- **IPC:** Se trata del número de instrucciones ejecutadas por ciclo.
- **LLCRPKI:** Este mide el número de referencias a la cache de tercer nivel por instrucción.

Una vez tenemos nuestros contadores y métricas definidas solo queda preparar la configuración de contadores que será expuesta a PMCTrack. Para esto tenemos que definir la estructura *pmcsched_counter_config_t*.

```
static pmcsched_counter_config_t cconfig = {
.counter_usage = {
    .hwpmc_mask = 0x0ff,
    .nr_virtual_counters = CMT_MAX_EVENTS,
    .nr_experiments = 1,
    .vcounter_desc = {"llc_usage", "total_llc_bw", "local_llc_bw"},
},
.pmcs_descr = &pmc_configuration,
.metric_descr = {&metric_description, NULL},
.profiling_mode = TBS_SCHED_MODE,
};
```

Como podemos observar en este struct también nos permite definir contadores virtuales y también el modo de *profiling* el cual se usa para definir cuando se llama a la *callback sched_timer_periodic*, puede ser de dos tipos:

- **TBS_SCHED_MODE:** Este modo funciona a través de un timer el cual podemos definir su tiempo al cual se le llama periódicamente a la *callback*.
- **EBS_SCHED_MODE:** Este modo funciona sobre eventos, en este se define una venta de eventos (ebs_windows), en vez de ejecuciones periódicas, este al estar dentro de esa ventana de eventos si algún evento ocurre se activará la *callback*.

Con todo esto ya solo queda definir en nuestro *plugin* las funciones que este implementa y darle un ID que lo identifique de cara a agregar este a PMCSched.

```
sched_ops_t example_plugin = {
      .policy = SCHED_EXAMPLE_MM,
      .description = "App classification plugin",
      .flags = PMCSCHED_CPUGROUP_LOCK,
      .counter_config = &cconfig,,
      .sched_timer_periodic = sched_timer_periodic_classification,
      .on_active_thread = on_active_thread_classification,
      .on_inactive_thread = on_inactive_thread_classification,
      .on_exit_thread = on_exit_thread_classification
};
```

Para agregar nuestro *plugin* a PMCSched, primero debemos definir la estructura de operaciones del *plugin* que acabamos de definir como extern dentro de *pmcshed.h*.

```
extern struct sched_ops dummy_plugin;
extern struct sched_ops group_plugin;
extern struct sched_ops busybcs_plugin;
extern struct sched_ops example_plugin;
```

Además deberemos de colocar el ID que previamente asignamos a este dentro del enum de políticas de *pmcshed.h*.

```
typedef enum {
    SCHED_DUMMY_MM=0,
    SCHED_GROUP_MM,
    SCHED_BUSYBCS_MM,
    SCHED_EXAMPLE_MM,
    NUM_SCHEDULERS
} sched_policy_mm_t;
```

Para terminar, se añade un puntero a nuestra estructura de operaciones del *plugin*, dentro de la estructura de *plugins* disponibles de PMCSched.

```
static  __attribute__ ((unused)) struct sched_ops*
available_schedulers[NUM_SCHEDULERS]= {
    &dummy_plugin,
    &group_plugin,
    &busybcs_plugin,
    &example_plugin,
 }
```

Tras esto solo queda añadir el nuevo *plugin* al Makefile del modelo del kernel específico de nuestra arquitectura, siendo este *src/modules/pmcs/intel-core/Makefile* si se trata de un *plugin* para sistemas con procesador Intel.

```
MODULE_NAME=mchw_intel_core
obj-m += $(MODULE_NAME).o
CONFIDENTIAL=1
PMCSCHED-objs= pmcsched.o dummy_plugin.o group_plugin.o busybcs_plugin.o
example_plugin.o
```

2.2. Het-Harness

Het-Harness es una herramienta formada por un conjunto de scripts, programas C/C++, librerías y ficheros de configuración. Fue diseñado para facilitar la evaluación experimental de algoritmos de planificación a nivel de sistema operativo y funciona en la mayoría de sistemas operativos tipo UNIX, como GNU/Linux, Solaris o Mac OS. Actualmente Het-Harness no es una herramienta open source. [2]

Hemos utilizado esta herramienta para ayudarnos en el lanzamiento de múltiples aplicaciones en sistemas multicore, creando scripts que lanzan conjuntos de experimentos automáticamente y generan ficheros log de salida con los resultados de los experimentos.

Este framework se basa en dos abstracciones básicas: *benchmark* y *benchset*. Un *benchmark* es una aplicación que puede ser lanzada como una parte de una carga de trabajo (conjunto de aplicaciones ejecutadas simultáneamente). Un *benchset* es una secuencia de cargas de trabajo, es decir, está compuesto por cargas de trabajo que se lanzan una tras otra. Para lanzar cualquier aplicación con este framework hay que tener registrado esta aplicación como *benchmark* en el fichero *$HET_HARNESS_ROOT/etc/harness/benchmark_list* y tener un script específico para su lanzamiento.

A continuación se muestra un ejemplo de cómo es un fichero *benchmark_list*, en el que asignamos a las variables los directorios que nos llevan a los scripts específicos de cada *benchmark*:

```
# SPECCPU 2017
BLENDER17   benchmarks/common/speccpu2017/blender17
BWAVES17  benchmarks/common/speccpu2017/bwaves17
CACTUBSSN17  benchmarks/common/speccpu2017/cactuBSSN17
CAM417          benchmarks/common/speccpu2017/cam417
DEEPSJENG17  benchmarks/common/speccpu2017/deepsjeng17
EXCHANGE217  benchmarks/common/speccpu2017/exchange217
FOTONIK3D17  benchmarks/common/speccpu2017/fotonik3d17
GCC17       benchmarks/common/speccpu2017/gcc17
IMAGICK17    benchmarks/common/speccpu2017/imagick17
```

[2] Het-Harness todavía no tiene documentación pública que poder citar, y se usa actualmente de forma interna en el grupo de ArTeCS de la UCM [4]

Y el siguiente fichero es un ejemplo de *benchset*:

```
#Benchmark List
gamess06=(GAMESS06)
soplex06=(SOPLEX06)
h264ref06=(H264REF06)
gobmk06=(GOBMK06)
equake4=(EQUAKE_M 4 spin "dynamic,30")
fftw3d4=(FFTW3D 4)
ep_npb4=(EP_NPB 4 spin "dynamic,30")
#

#Experiments
exp1=(gamess06 soplex06 h264ref06 gobmk06)
exp2=(fftw3d4)
exp3=(equake4)
exp4=(ep_npb4)
#Test Vector
test_vector=(exp1 exp2 exp3 exp4)
```

Como podemos observar en el ejemplo de *benchset* anterior, un archivo *benchset* está compuesto por 3 partes:

- Descripción de línea de comando que se usará para ejecutar el *benchmark*, donde el primer parámetro es el ID del *benchmark* y los siguientes son los parámetros que se pasarán al script correspondiente. En el ejemplo vemos que GAMESS06 o SOPLEX06 no tienen parámetros que pasar, sin embargo, EQUAKE_M por ejemplo tiene 3 parámetros: 4, spin y "dinamic,30".

- La segunda parte indica de qué *benchmarks* está compuesto las cargas de trabajo, cada elemento del array es una variable que hemos declarado anteriormente y está asociado a un *benchmark*. Podemos observar que tenemos cuatro cargas de trabajo, donde la primera esta formado por cuatro aplicaciones y el resto solamente tiene una aplcación.

- La tercera parte es el array *test_vector*, donde los elementos internos deben coincidir con los workloads anteriores.

2.2.1. Arquitectura de Het-Harness

La arquitectura de Het-Harness se compone de cuatro capas de alto nivel. Como se muestra en la figura 2.2, el nivel 0 contiene los scripts de los *benchmarks*. En el nivel 1 se encuentran las cargas de trabajo y el lanzador escrito en C++. El nivel 2 incluye los benchsets y, finalmente, en el nivel 3 se realiza el lanzamiento de un *benchset* con diferentes configuraciones.

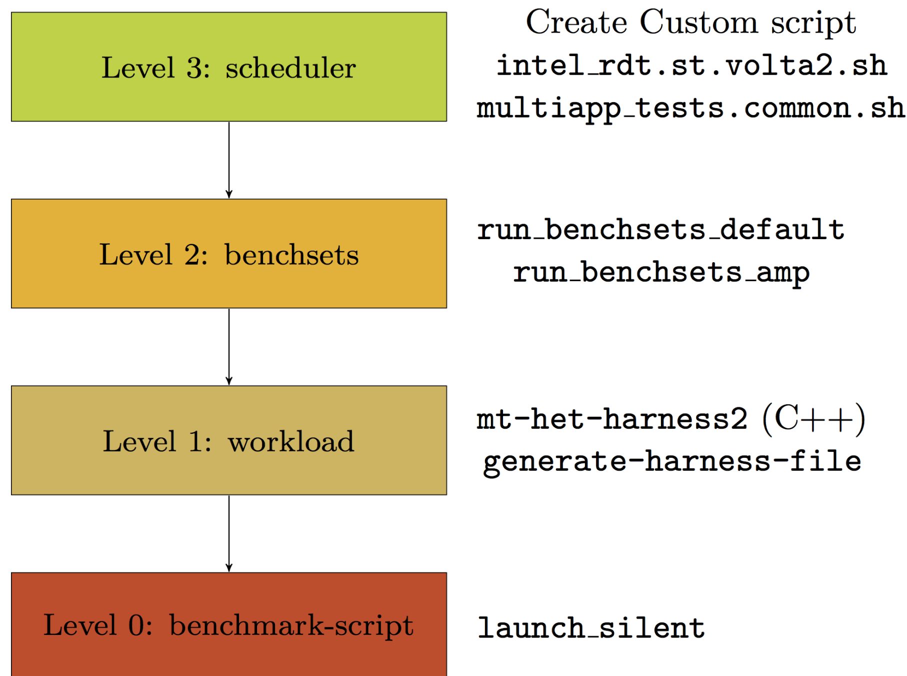


Figura 2.2: Arquitectura de Het-Harness

- Capa 0: en esta capa incluimos todos los scripts de *benchmarks* (entre ellos se encuentra el script genérico *launch.silent* que reacciona a las variables de entorno y establece configuraciones en la máquina) que se lanzan como parte de las cargas de trabajo. Es importante notar que, dentro de Het-Harness, no se incluyen los archivos ejecutables ni los archivos de entrada de datos de los *benchmarks*. Estos deben almacenarse por separado en el sistema, en un directorio que contenga los ejecutables y los datos de entrada para cada *benchmark*.

- Capa 1: en este nivel se definen y ejecutan las cargas de trabajo mediante el lanzador del framework (que se encuentra también es esta capa), escrito en lenguaje C++. Este lanzador inicia una sola carga de trabajo, compuesta por un conjunto de aplicaciones, con una configuración fija del sistema.

- Capa 2: es la capa donde se gestionan cada *benchset* completo, que es un array de cargas de trabajo, también con una configuración fija del sistema. Para lanzar un *benchset* con el lanzador se hace de esta forma:

```
<launcher> <benchset_file.spec> <configuration_file>
```

  El fichero de configuración es un script bash que define un número de variables por defecto y las variables de entorno.

- Capa 3: este nivel es el que gestiona el lanzamiento de un *benchset* pero con múltiples configuraciones. Para hacer esto posible, incluimos el script de bash común en nuestro propio script, el cual se encuentra en:

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

Este script de bash tiene las funciones personalizadas para establecer las diversas configuraciones del sistema asociadas con cada algoritmo de planificación considerado en los experimentos. Y para cada configuración del sistema hay que definir nuestras propias funciones bash de esta forma:

```
function scheduler1(){
 export MY_COOL_ENVAR=3
 activate_scheduler 5
  ... set up desired HARNESS environment variables for the test....
  ... set up scheduler parameters via /proc ...
 export tag=".scheduler1"
}

function scheduler3(){
  ... set up desired HARNESS environment variables for the test....
  ... set up scheduler parameters via /proc ...
 export tag=".scheduler3"
}
```

Definir también el array de experimentos que queremos lanzar y las variables con los ficheros de configuración y de benchset que vamos a utilizar para el lanzamiento.

```
experiments=(stock_linux scheduler1 scheduler2 scheduler3)
 cfg_file="${HET_HARNESS_ROOT}/test_scripts/intel-cmt/ \
 intel-cat-user.niter.cfg"
 spec_files=(${HET_HARNESS_ROOT}/benchsets/intel-cat/phases_8_apps.spec)
```

Por último definimos una función para establecer el método de limpiar las variables de entorno y preparar las configuraciones entre los experimentos:

```
function restore_defaults_custom()
{
    restore_defaults
    export HARNESS_RANGE="1" ## Launch only workload 1 of each
        benchset
    ## Override default timeout and number of samples of config file
    export HARNESS_SAMPLES=3
    export HARNESS_TIMEOUT=100
}
```

La variable *HARNESS_RANGE* define el rango de cargas de trabajo que se lanzarán del conjunto de *benchmarks*; en este caso, solo se lanzará la carga de trabajo 1 del conjunto. *HARNESS_SAMPLE* indica cuántas instancias del *benchmark* deben ejecutarse, y *HARNESS_TIMEOUT* especifica el tiempo mínimo que una aplicación debe estar en ejecución.

2.3. Cloudsuite

CloudSuite es un conjunto de *benchmarks* diseñado para evaluar el rendimiento de servicios en la nube en distintas arquitecturas. Consta de un total de 8 *benchmarks* que representan servicios en línea populares y cargas de trabajo analíticas en centros de datos [1]. En este proyecto, hemos estado trabajando con la versión 4.0 de esta herramienta, donde se ha incluido una actualización completa de la pila de *software* y correcciones de errores para todas las cargas de trabajo.

Los *benchmarks* se basan en pilas de software de código abierto y requieren la instalación de Docker para su utilización, ya que las aplicaciones se ejecutan en contenedores para simplificar su uso. Dentro de los *benchmarks*, encontramos una variedad de tipos, que van desde aquellos que simulan un estado representativo de una base de datos NoSQL en la nube, hasta los que implementan algoritmos de filtrado colaborativo en conjuntos de datos de valoraciones de usuarios sobre películas.

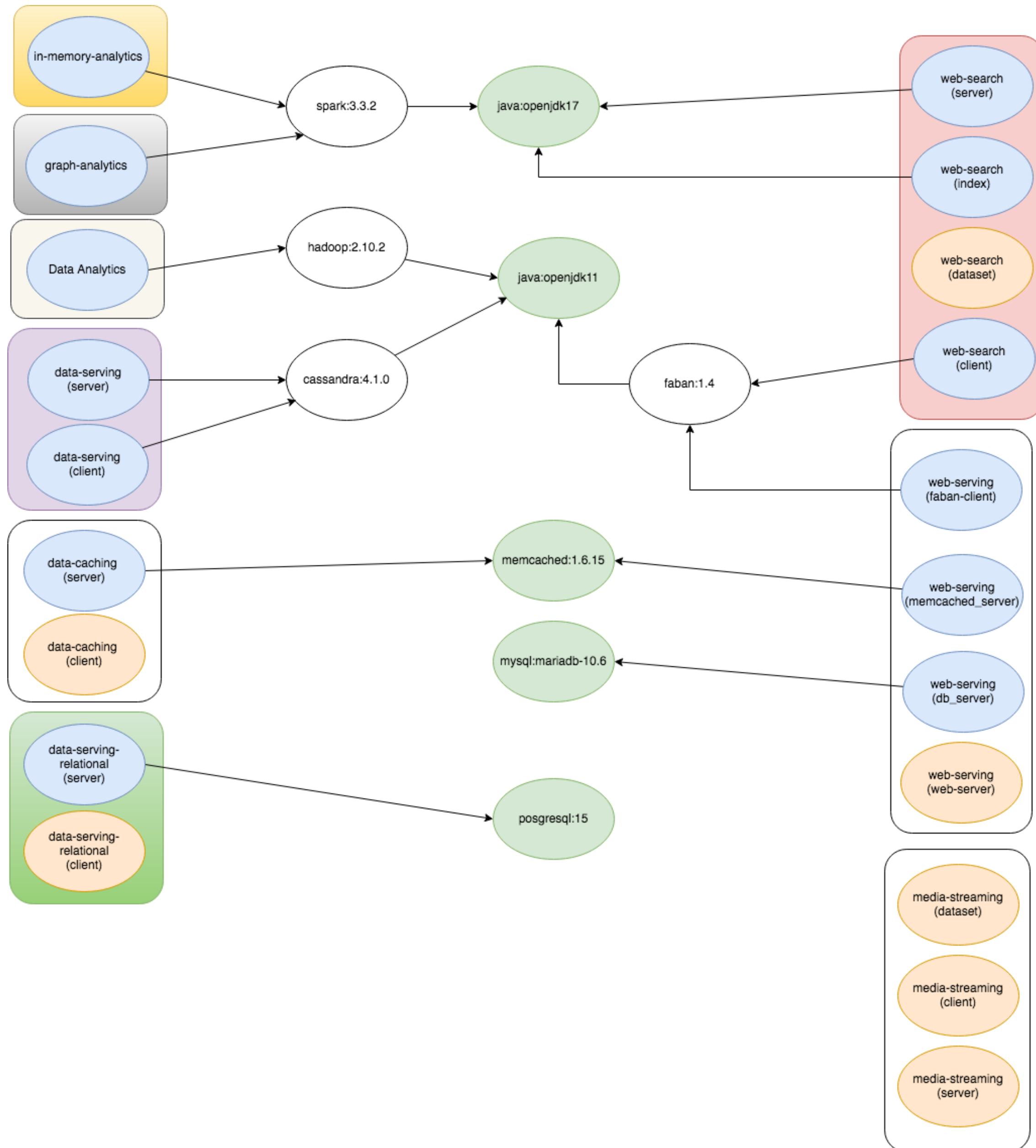


Figura 2.3: Diagrama de dependencias de los *benchmarks* de Cloudsuite

En la figura 2.3 se muestran los distintos *benchmarks* de CloudSuite y las dependencias entre los distintos contenedores que constan. Cada círculo representa un contenedor Docker, mientras que cada caja representa un *benchmark*. Los contenedores naranjas dependen únicamente del contenedor básico de CloudSuite (que es un contenedor Ubuntu 22.04). Por otro lado, los contenedores verdes también dependen del contenedor básico de CloudSuite, pero son utilizados por otras aplicaciones para construir su propio contenedor. Finalmente, los contenedores blancos representan contenedores intermedios que dependen de los contenedores verdes y sirven como base para algunos de los contenedores de aplicaciones.

Además de las dependencias mostradas en la figura, es importante destacar que esta no muesta los conjuntos de datos utilizados por algunos *benchmarks* por sim-

plicidad. Por ejemplo, el *benchmark* de Data Caching requiere el uso de un conjunto de datos de Twitter (ahora denominado X), lo que implica la necesidad de crear un contenedor específico también para cada conjunto de datos utilizado por los *benchmarks*.

Durante la exploración y uso de esta herramienta, hemos identificado algunos problemas. Por ejemplo, notamos que los *benchmarks* se basan en imagenes Docker que se descargan de la nube de Cloudsuite directamente [8], y esas imágenes no están bajo nuestro control. Por lo tanto, necesitamos encontrar una forma de generar los contenedores localmente para asegurar un entorno controlado. Además, enfrentamos problemas de ralentización al ejecutar algunos *benchmarks* en nuestro entorno de desarrollo, lo que demandará modificaciones en el código del *benchmark* para solucionarlo. En la sección 5.2.2 se puede encontrar las descripciones de las distintas *benchmarks* que tiene Cloudsuite.

2.3.1. Adaptaciones de Cloudsuite

Para solucionar los distintos problemas que se nos presentaron, aplicamos las siguientes soluciones:

- Para abordar el problema de descargar imágenes desde la nube de CloudSuite para la creación de contenedores durante la ejecución de los *benchmarks*, creamos un repositorio propio (*fork* del original) donde almacenamos la versión 4.0 de CloudSuite. Posteriormente, ajustamos los *dockerfiles* para que empleen las imágenes locales en lugar de descargarlas desde la nube. Para estandarizar los nombres de las imágenes, añadimos el prefijo *'cs4-'* a cada nombre de imagen generado. En la figura 2.4, podemos ver el Dockerfile adaptado para el servidor del *benchmark* de ejemplo de Data Serving Relational. Este Dockerfile crea la imagen del servidor basándose en la imagen local *'cs4-postgresql:15'*. Además, instala los paquetes necesarios de Python, copia los scripts necesarios al contenedor y define el script *docker-entrypoint.py* como el punto de entrada del contenedor.

```
FROM cs4-postgresql:15

# Install sudo for user switching
RUN apt update && apt install sudo python3 -y

# Make the database access public
RUN echo 'host\tall\tcloudsuite\t0.0.0.0/0\tscram-sha-256' >> \
/etc/postgresql/15/main/pg_hba.conf

# Copy the entrypoint
COPY ./docker-entrypoint-local.py /root/docker-entrypoint.py

RUN chmod +x /root/docker-entrypoint.py

## Script for db relocation
ADD change-database-location.sh /change-database-location.sh
RUN chown root:root /change-database-location.sh
RUN chmod 700 /change-database-location.sh

ENTRYPOINT ["/root/docker-entrypoint.py"]
```

Figura 2.4: *Dockerfile* adaptado del servidor de Data serving relational

- El siguiente inconveniente que nos enfrentamos es la ralentización de la ejecución de algunos *benchmarks* como el de Data Serving Relational. Tras probar con otro equipo, descubrimos que esto se debe a que en nuestro entorno de desarrollo, el directorio raíz está instalado en un disco magnético, lo que ralentiza considerablemente la ejecución del *benchmark*. La solución que implementamos fue cambiar la ruta de acceso al conjunto de datos para que apunte a una ruta ubicada en un disco de estado sólido del entorno que estamos utilizando.

- En este punto, en lugar de abordar un problema, vamos a describir una extensión que hemos realizado en la herramienta. Hemos añadido un nuevo *benchmark* al conjunto existente, denominado Data Serving MySQL, el cual se deriva de Data Serving Relational. La diferencia principal esta en el uso de MySQL en lugar de PostgreSQL como sistema de gestión de base de datos. Para implementar esta extensión, creamos todos los archivos necesarios, incluidos los Dockerfiles para el cliente y el servidor, así como los scripts de entrada para los contenedores. Luego, configuramos la extensión en nuestro entorno para garantizar su funcionamiento.

Capítulo 3

Adaptaciones en Het-Harness

En este capítulo detallaremos las modificaciones implementadas en la herramienta Het-Harness para satisfacer los requisitos de nuestro proyecto. Antes de la actualización, Het-Harness solo tenía la capacidad de lanzar aplicaciones de un solo proceso. En la figura 3.1 se ilustra el flujo de ejecución de una carga de trabajo de ejemplo con tres aplicaciones. Inicia en la capa de cargas de trabajo (Multi-application workload), tras lo cual lanzador C++ (capa 1) ejecuta los scripts de las tres aplicaciones. Cada una de estas aplicaciones utiliza el script *launch.silent*, que responde a las variables de entorno para configurar la ejecución de la aplicación, como el uso de PMCTrack o el particionado de cache.

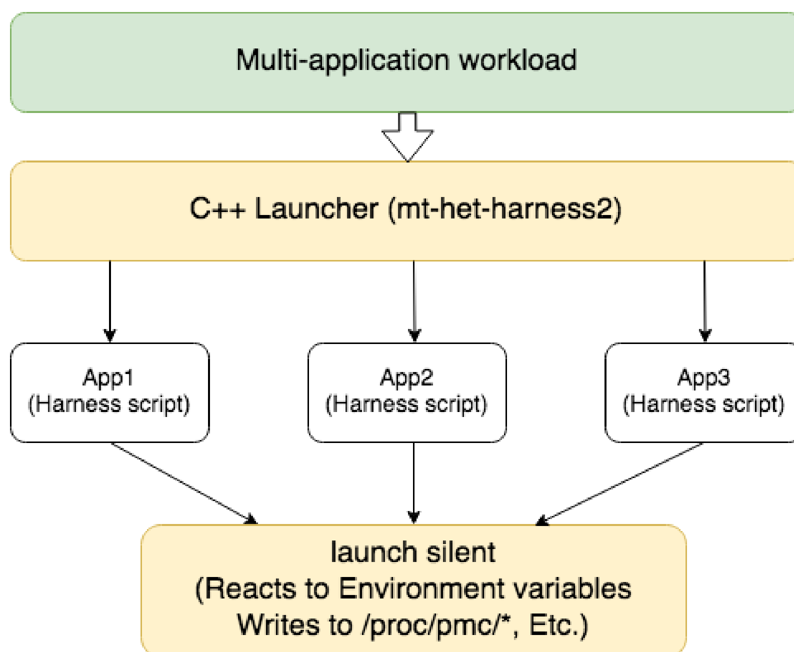


Figura 3.1: Flujo de la carga de trabajo multi-aplicación antes de la actualización

Para cubrir las necesidades de nuestro proyecto, hemos llevado a cabo las modificaciones en dos fases:

- **Fase 1, soporte de contenedores**: Basándonos en las capas previamente explicadas de esta herramienta (ver sección 2.2.1), se han realizado cambios significativos. En la Capa 0 , se han agregado nuevas variables de entorno para la configuración del *benchmark*. Respecto a la Capa 1, se ha modificado el lanzador C++ para permitir el lanzamiento de contenedores, cumpliendo con cuatro puntos principales: establecer las variables de entorno para configurar la ejecución del benchmark, iniciar el contenedor benchmarks, ejecutar internamente el contenedor y detener todos los procesos internos del contenedor y eliminarlo. Estos cuatro puntos fundamentales sirvieron como base para el desarrollo del comando *harness-docker*, que nos permitirá el lanzamiento de aplicaciones en contenedores.

- **Fase 2, soporte para aplicaciones Cloudsuite**: Se ha incorporado un nuevo tipo de scripts a la herramienta para solucionar el problema que presentaba la versión anterior, que no permitía lanzar varios contenedores para un mismo benchmark.

3.1. Soporte inicial de contenedores

Basándonos en los cuatro puntos principales mencionados anteriormente, utilizamos *harness-docker* para la gestión de contenedores, que es esencialmente un script que ejecuta el comando *'harness-docker'* para administrar los contenedores, que tiene las siguientes opciones de ejecución:

- *buildenv <id_stream>* : Este comando crea un archivo con las variables de entorno necesarias para que *launch_silent* pueda realizar su función, que consiste en reaccionar a dichas variables de entorno y establecer las configuraciones apropiadas en la máquina. El parámetro *<id_stream>* es el ID que se le pasa a *harness-docker* para identificar el flujo de trabajo específico.

- *start <id_stream>*: inicia un contenedor con la imagen genérica de het-harness sin ejecutar nada.

- *exec <id_stream><harness_benchmark_script>*: Ejecuta el benchmark pasado por parámetro.

- *stop <id_stream>*: Detiene el contenedor correspondiente, y mata todos los procesos en ejecución internos del contenedor.

Se ha añadido una nueva funcionalidad para restringir las CPUs en que se ejecuta el contenedor, con una nueva variable de entorno *HARNESS_DOCKER_CPUSET* que se usa junto a *harness-docker* cuando se lanza el contenedor. Además, en esta versión de Het-Harness, se pueden lanzar las aplicaciones existentes y alojarlas en contenedores independientes. En la figura 3.2 se presenta el flujo de ejecución de una carga de trabajo compuesta por tres aplicaciones, que hace uso del lanzador

C++ y el comando *harness-docker*. Para poder lanzar los contenedores, es necesario instalar una imagen Docker genérica llamada *het-harness*, ya que estos parten de dicha imagen. Los contenedores ejecutan los scripts de las aplicaciones y utilizan el script *launch.silent*, que responde a las variables de entorno para configurar la ejecución de la aplicación.

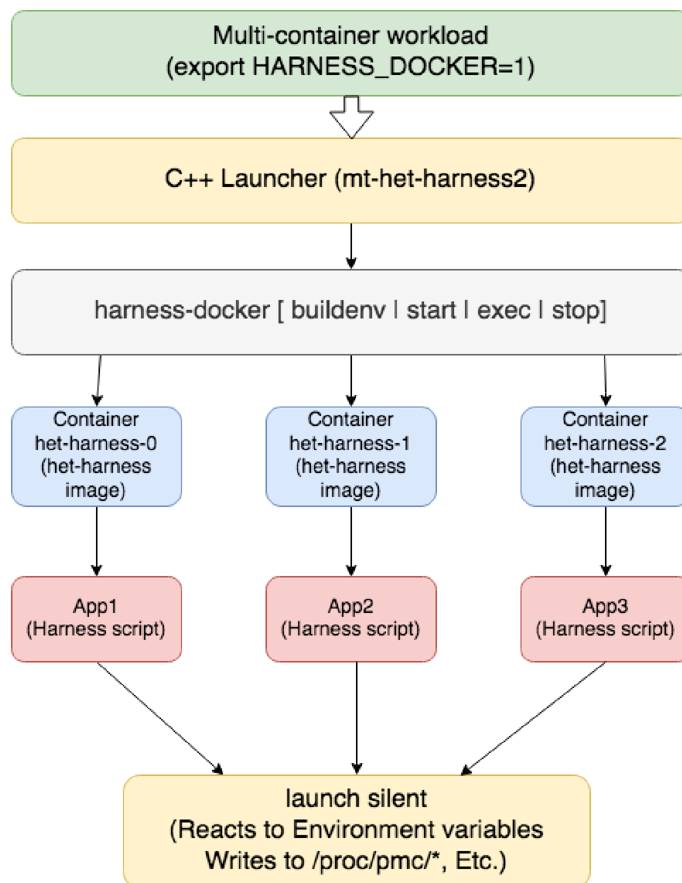


Figura 3.2: Flujo de la carga de trabajo multi-contenedor con aplicaciones convencionales

Pero sigue habiendo problemas en esta versión inicial:

- No se puede hacer uso de los contenedores *custom* que crea Cloudsuite internamente, y algunos de los *benchmarks* de Cloudsuite necesitan varios contenedores por *benchmark*.
- Tampoco se puede hacer uso de PMCTrack ya que no hay acceso a */proc/pmc*, porque es un directorio protegido que el contenedor no puede acceder.
- Hay *benchmarks* de Cloudsuite donde se crean varios contenedores, por lo que se trata de un conjunto de procesos en vez de una sola aplicación multi-hilo, es decir, hay aplicaciones multi-contenedor.

3.2. Soporte para aplicaciones Cloudsuite

En la segunda fase de la actualización de Het-Harness, ya es posible lanzar los *benchmarks* de Cloudsuite utilizando un nuevo tipo de script incorporado a Het-Harness. En la figura 3.3 se ilustra el flujo de ejecución de una carga de trabajo compuesta por una aplicación existente y dos nuevas aplicaciones de Cloudsuite que requieren el nuevo script para gestionar los contenedores necesarios para el *benchmark*. Estos scripts son capaces de lanzar tantos contenedores como sea necesario. No es necesario invocar manualmente estos scripts; en su lugar, se utiliza el comando *harness-docker*, modificando la variable de entorno *BENCHNAME* donde se guarda el nombre del *benchmark*. Finalmente, estos contenedores ejecutan los scripts de las aplicaciones.

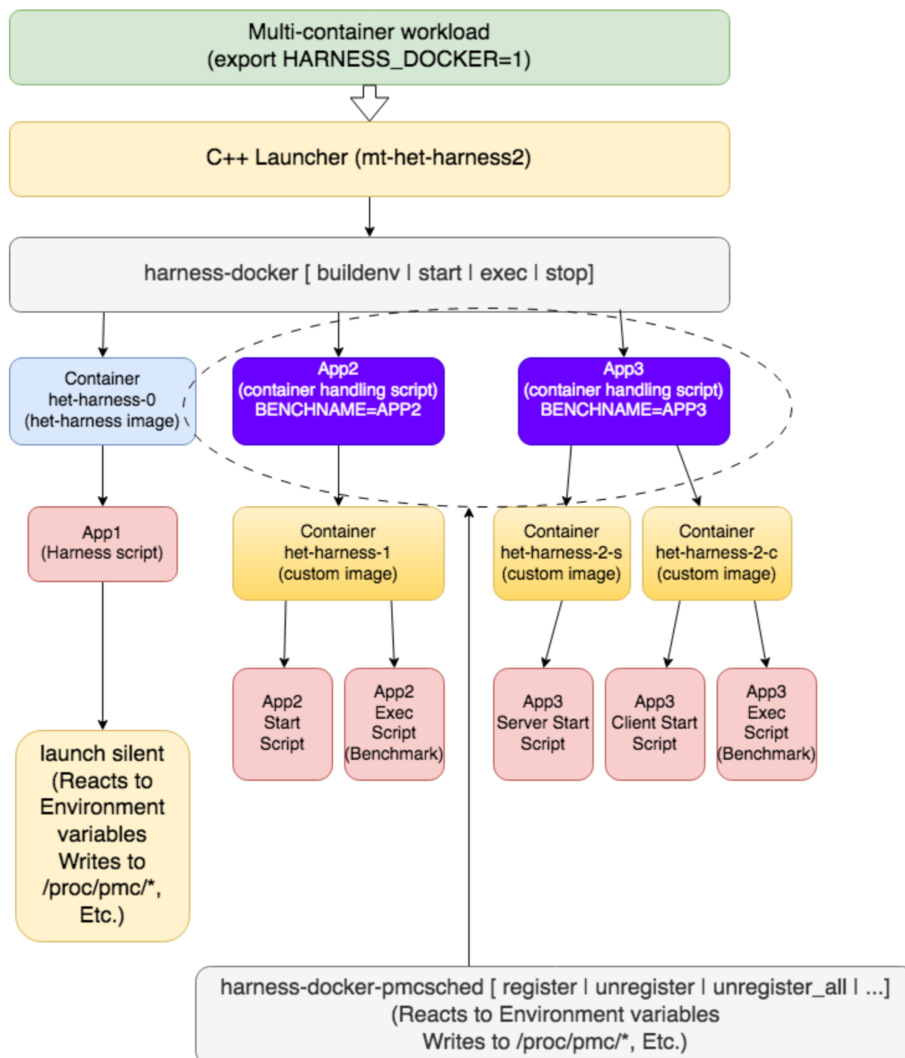


Figura 3.3: Flujo de la carga de trabajo multi-contenedor con aplicaciones de Cloudsuite

Los puntos principales de la actualización son los siguientes:

- Los scripts deben darse de alta en el fichero de *benchmark_list* y *container_list* para su correcto funcionamiento, ya que *harness-docker* busca en estos ficheros el *BENCHNAME* que se le pasa.

- Las opciones de *harness-docker* son las mismas, aunque sí han cambiado su modo de funcionamiento:

  - *start*: Inicia todos los contenedores necesarios, limita los CPUs que pueden ser utilizados por el/los contenedores según la variable de entorno *HARNESS_DOCKER_CPUSET*, y con la variable de entorno *HARNESS_PMCTRACK* se determina si es visible para los plugins de PMCTrack/PMCSched. En esta opción, los benchmarks de cliente-servidor ejecutan el proceso de *warmup* para *'calentar'* el servidor. Podemos ver un ejemplo de ejecución con el siguiente comando, que arranca el benchmark In Memory Analytics, restringe las CPUs 8-15 y lo inicia con el número de flujo 0:

    ```
    BENCHNAME=IN_MEMORY_ANALYTICS_CS4 \
    HARNESS_DOCKER_CPUSET=8-15 harness-docker start 0
    ```

  - *exec*: Ejecuta el benchmark en el/los contenedores necesarios, si la variable de entorno *RAW=yes* entonces saca la salida del benchmark por la salida estándar.
  - *stop*: Detiene todos los contenedores lanzados anteriormente.

3.3. Sistema de scripting

Con la integración de Cloudsuite en la herramienta, se introdujo el comando *harness-docker* para la gestión de contenedores. Esta adición requiere la implementación de scripts de benchmarks que ejecuten las distintas opciones admitidas por este comando, como se explicó anteriormente: *start*, *exec*, *stop*, así como nuevas opciones que se agregan según el *benchmark*. Por ejemplo, la opción *max-tps* que se añade a los *benchmarks* cliente-servidor para establecer el máximo TPS (Transactions Per Second, que es una métrica utilizada para medir el rendimiento de un sistema en términos de la cantidad de transacciones que puede procesar por segundo) durante su ejecución.

3.3.1. Ejemplo de un *benchmark script*, de Cloudsuite

El siguiente ejemplo es un script del *benchmark* de ejemplo, el cual se basa en el framework Spark [7]:

- En la figura 3.4 podemos ver el comienzo del script, donde se definen las variables necesarias. Estas incluyen la acción a realizar (*start*, *stop*, etc.), los

nombres de los contenedores que se crearán, la imagen en la que se basarán los contenedores (que debe estar descargada localmente), y los archivos que se generarán durante la ejecución de los contenedores. En caso excepcional, se pasa un tercer parámetro, ya que este *benchmark* tiene tres modos correspondientes a los tres algoritmos que puede ejecutar, y estos deben pasarse como parámetro.

```
#!/bin/bash
action=${1:-help}
stream_id=${2:-0}
## The benchmark supports three workload names
# PageRank (pr)
# Connected components (cc)
# Triangle count (tc)
workload_name=${3:-"tc"}
container_name="het-harness-${stream_id}"
## Not sure if envfile will be used
envfile="/tmp/$USER/envfile-${stream_id}"
#discard first three arguments
shift ; shift ; shift
args="$*"
exec_command="/root/entrypoint.sh --driver-memory 8g \
--executor-memory 8g ${args}"
image_name="cs4-graph-analytics"
presence_file=/tmp/${USER}/${image_name}.container
pmcsched_file=/tmp/${USER}/${image_name}.pmcsched
...
```

Figura 3.4: Las variables del inicio del script del *benchmark* de ejemplo

- Después, entramos en casos según el valor de la acción. Como se muestra en la figura 3.5, que presenta la sección del script donde procesamos la acción *start*, ejecutamos el comando *docker container run* para crear los contenedores necesarios, pasándole el nombre de contenedor y de imagen declarados anteriormente. También podemos incluir la variable *binding_spec*, que restringe los CPUs que los contenedores pueden utilizar. Finalmente, establecemos un estado llamado *sleep infinity*, donde los contenedores permanecerán inactivos. En la parte final del script, se utiliza la variable de entorno *HARNESS_PMCTRACK* para verificar si está activada y registrar este *benchmark*. Si se trata de un *benchmark* de cliente-servidor, también habrá una etapa de calentamiento (*warmup*) dentro de *start*.

```
...
binding_spec=""

## Add support for afinitization
if [ "${HARNESS_DOCKER_CPUSET}" != "" ]; then
    binding_spec="--cpuset-cpus=${HARNESS_DOCKER_CPUSET}"
fi

## Start
docker container run --rm -d --name ${container_name} \
-h ${container_name} ${binding_spec} \
--entrypoint "" \
--volumes-from twitter-data --cap-add SYS_NICE \
${image_name} sleep infinity > ${presence_file}


## Register in PMCsched if needed
if [ "${HARNESS_PMCTRACK}" == "yes" ]; then
    harness-docker-pmcsched register ${stream_id} > ${pmcsched_file}
fi
...
```

Figura 3.5: La opción de *start* del script del *benchmark* de ejemplo

- Como se muestra en la figura 3.6, que es parte del script donde procesamos la opción de *exec*. Primero verificamos si la ejecución es interactiva, es decir, si se mantiene abierta la entrada estándar del contenedor para permitir la interacción, utilizando la variable *interactive*. Y después ejecutamos el contenedor utilizando el dataset *'twitter-data'*. Además, comprobamos al final si la variable *RAW=yes* está configurada para determinar si se deben mostrar los resultados del *benchmark* por la salida estándar.

```
...
if [ -c "${ttyfile}" ] ; then
    interactive="-it"
fi


command="docker container exec ${interactive} \
-e WORKLOAD_NAME="${workload_name}" \
${container_name} time -p ${EXEC_PREFIX} ${exec_command}"

if [ "$DEBUG" == "yes" ]; then
    echo $command
fi

if [ "$RAW" == "yes" ]; then
    exec $command
else
    exec $command > /dev/null 2>&1
fi
...
```

Figura 3.6: La opción de *exec* del script del *benchmark* de ejemplo

- Finalmente, la opción *stop*, como se muestra en la figura 3.7, consiste en dos comandos. Uno detiene todos los contenedores creados anteriormente junto con sus procesos internos, y el otro elimina los archivos generados junto a estos contenedores.

```
...
docker container stop -t 2 ${container_name}

rm ${presence_file}
...
```

Figura 3.7: La opción de *stop* del script del *benchmark* de ejemplo

Capítulo 4

Adaptaciones de PMCSched

Para el uso de PMCSched con *benchmarks* de CloudSuite, este presentaba un grave problema, los *benchmarks* de CloudSuite se lanzan dentro de contenedores Docker. El framework PMCSched pese a poder ser usado con contenedores, este no presenta soporte específico para el uso de contenedores, lo que no nos gestionar todos los procesos de un contenedor dentro de un *plugin* como un todo.

Los contenedores pueden tener varias aplicaciones ejecutándose dentro de estos y cada una de estas lanzando sus propios hilos. A la hora de desarrollar nuestros *plugins*, se necesita poder tener una manera de identificar a que contenedor pertenece cada hilo, para que los *plugins* puedan seguir haciendo uso de este framework para la gestión de recursos y de cache entre los distintos procesos.

4.1. Contenedores como aplicación multihilo

Debido al problema de no poder asociar los hilos que se crean a través de un contenedor a este por parte de PMCSched, se tuvieron que realizar modificaciones en el framework. Para lograr esto, se adapto el framework de PMCSched de tal forma que los contenedores en la práctica se exponen como una aplicación multihilo.

Para poder identificar los hilos que pertenecen a un mismo contenedor, se hace uso de la abstracción del kernel de Linux de los *cgroups*. Cuando Docker crea un contenedor, este crea un cgroup asociado a este, todos los procesos asociados a un contenedor forman parte de ese cgroup.

De esta forma todos los hilos que se creen dentro de un contenedor serán tratados por PMCSched de tal forma a como se hace con las aplicaciones multihilo. Esto se traduce a que de cara a los *plugins* que se creen dentro de PMCSched los contenedores serán una aplicación multihilo.

Con estas adaptaciones en el framework, también se pretende poder realizar

particionado de cache y moniotrización del consumo de ancho de banda de los contenedores. Por esto es necesario introducir previamente los conceptos del RMID Y CLOS ID. El RMID es un identificador de monitorización, perteneciente a las tecnologías de Intel CMT (Cache monitoribg TECH) y Intel MBM (Memory Bandwidth monitoring), el cual nos permite monitorizar el uso de LLC y ancho de banda. El CLOS ID (Class of Service) se trata de otro identificador preteneciente a la tecnología Intel CAT (Cache Allocation Technology)[5], el cual nos permite la particion configurable de la LLC a través de máscaras de bits.

```
typedef struct sched_app {
    atomic_t ref_counter;
    rwlock_t app_lock; /* For global app fields */
    app_t_pmcsched pmc_sched_apps[MAX_GROUPS_PLATFORM];
    unsigned char is_multithreaded;
    schedctl_t* master_shared; /* Shared memory region master thread */
    schedctl_notifier_t master_notifier;
    struct task_struct* schedctl_signal_recipient;
    pid_t pid; /* To keep track of the process ID */
    struct task_struct* leader; /* This may be NULL when using containers */
    struct css_set* cgroups;     /**
                                 * To point to container
                                 * (if this is representing one )
                                 **/
    struct list_head cgroup_node; /* Linkage for container list */
    container_properties_t cprops;
    app_socket_stats_t socket_stats[MAX_SOCKETS_PLATFORM];
} sched_app_t;
```

Para poder exponer como aplicaciones multihilo los contenedores, se añadieron tres nuevos campos a la estructura *sched_app_t* que define las aplicaciones en PMCSched. Los nuevos campos son:

- **cgroups:** Este campo almacena la información del *cgroup* que representa esta aplicación en caso de tratarse de un contenedor. Además si esta aplicación no se trata de un contenedor el valor de este puntero será *null*.
- **cgroup_node:** Este sirve para poder enlazar el contenedor a la lista enlazada de los distintos contenedores que se encuentran registrados en PMCSched. Al igual que pasa con el campo *cgroups*, no se encontrará asociado a ninguna lista enlazada en el caso de que la aplicación correspondiente no se trate de un contenedor.
- **cprops:** Este campos se trata de una estructura encargada de guardar información relativa a los contenedores. En esta encontramos el valor de su *cgroup*, en caso de no ser un contenedor el valor de este será menos uno. También encontramos un campo para guardar su *CLOS* (Class of service), el cual puede ser especificado por el usuario, si este no ha sido definido por el usuario su valor será cero.

```
typedef struct container_props {
    int clos_id; /* User-defined Class of Service. Default is 0 */
    int container_group; /* -1 if it does not belong to any group */
} container_properties_t;
```

Otro de los cambios que se han incorporado es la adición de un grupo de nuevas funciones dentro de PMCSched. Estas nos permiten registrar un nuevo contenedor cuando este se activa y así poder seguir la actividad de sus hilos o realizar el desregistro de un contenedor cuando este se desactiva. La función que se encarga del registro, inicializa las estructuras de memoria necesarias para el uso de contenedores y registra el nuevo contenedor en la lista de contenedores activas. A la hora de eliminar un contenedor registrado, la función implementada se encarga de liberar la memoria reservada para el contenedor y elimina el contenedor de la lista de contenedores activos.

```
/*****
 * Support for container activation
 **/
static sched_app_t* pmcsched_register_container(pid_t pid,
    container_properties_t* user_props );
static int pmcsched_unregister_container(pid_t pid);
```

Para terminar con las mejoras realizas, se han añadido dos nuevas *callbacks* a la interfaz de los *plugins* de PMCSched. Estas *callbacks* se llaman cuando se registra o elimina un contenedor y han ser usadas para la inicialización de los campos requeridos para el uso de contenedores por los distintos *plugins*.

```
/**
* Two callbacks added to manage
* container and plugin specific memory
* (e.g. a container is registered or its last
* active process or thread terminates)
**/
int (*on_new_container)(sched_app_t* capp, struct task_struct* container_parent);
void (*on_free_container)(sched_app_t* capp);
```

Por último, estas adaptaciones también nos permiten identificar a que contenedor pertenece un hilo haciendo uso del *RMID*, este es un ID asignado automáticamente para monitorizar el uso de espacio de la LLC y el uso de ancho de banda. El *RMID* es compartido por todos los hilos de un mismo contenedor.

Capítulo 5

Análisis Experimental

En esta sección se describe la plataforma experimental a través de la cual se han realizado los distintos experimentos. Además, se describe la motivación detrás de cada experimento, su realización y posterior análisis.

5.1. Plataforma Experimental

Durante el desarrollo del análisis experimental, hemos contado con varias plataformas experimentales las cuales han sido esenciales para el correcto desarrollo del análisis, la creación y lanzamiento de los distintos experimentos.

5.1.1. Plataforma Broadwell

Se trata de un servidor equipado con un procesador Intel de arquitectura Broadwell. Más concretamente es un *Intel(R) Xeon(R) CPU E5-2620 v4*, el cual trabaja a una frecuencia de 2.10GHz.

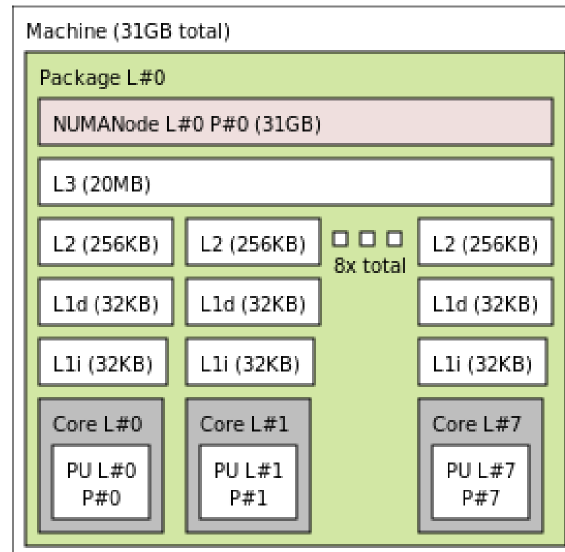


Figura 5.1: Topología del Intel(R) Xeon(R) CPU E5-2620 v4

La figura 5.1 describe la topología de la CPU equipada, notamos que este sistema tiene una única CPU la cual se divide en 8 cores. Esta CPU está equipada con una caché de tercer nivel de 20MB con 20 vías (L3), que es compartida entre todos los cores. Además cada núcleo cuenta con su propia caché de primer y segundo nivel, donde el segundo nivel de cache tiene una capacidad de 256KB. Este procesador tiene soporte de particionado de caché en la LLC (L3) mediante tecnología Intel CAT (*Cache Allocation Technology*) [5], la cual permite el particionado por vías de la cache con un particionado mínimo de hasta una vía.

5.1.2. Plataforma Skylake

Esta plataforma es el servidor usado para el lanzamiento de la mayor parte de los experimentos de este TFG. Este incorpora un procesador *Intel(R) Xeon(R) Gold 6138* (micro-arquitectura Skylake) el cual trabaja a una frecuencia de 2.00GHz.

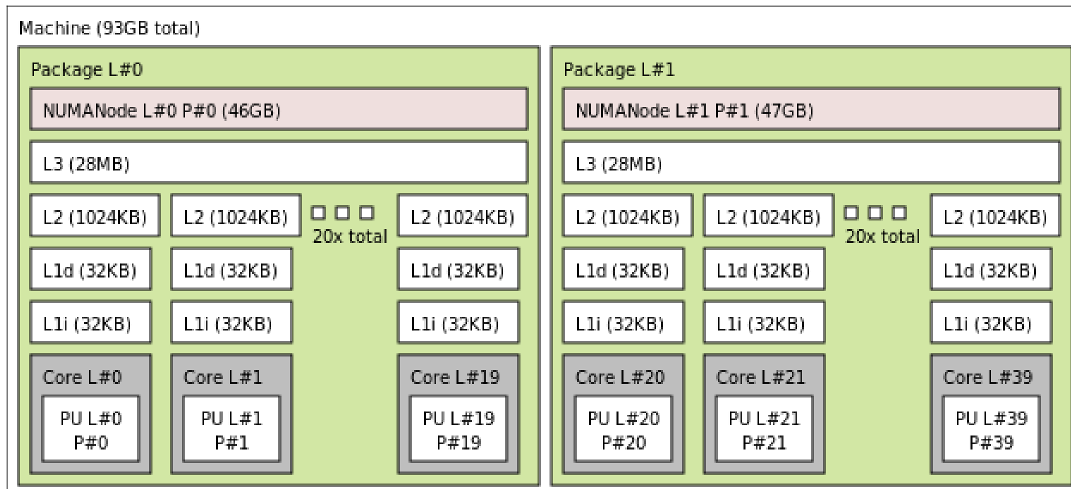


Figura 5.2: Topología del Intel(R) Xeon(R) Gold 6138

Como podemos observar, la figura 5.2 nos presenta la topología de las CPUs de este servidor. Esta plataforma posee dos CPUs, cada una de ellas integra 20 cores. Estas CPUs poseen cada una su propia cache de tercer nivel de 28MB con 11 vías de cache, que es compartida entre todos los cores del CPU. Cada core tiene su propia cache L1 y L2 de 1MB. Además estos procesadores también cuentan con la tecnología Intel CAT [5], para particionar la L3.

5.2. Benchmarks

Para llevar a cabo los distintos análisis experimentales realizados a lo largo del proyecto, ha sido necesario el uso de distintos tipos de *benchmarks*, cada uno de estos con distintas características a analizar y comparar entre sí.

5.2.1. HPC

Los benchmark HPC (*High Performance Computing*) son aplicaciones que representan el comportamiento de distintos tipos de programas usadas en el ámbito científico y de cómputo intensivo. Estos pueden ser aplicaciones de un solo hilo o multihilo y comportarse de maneras muy distintas en lo que respecta al uso de memoria, ancho de banda y latencia.

Se ha hecho uso de un gran número de estos *benchmarks* pertenecientes a distintas clases de *suites*. Especialmente, se ha considerado el uso de aplicaciones multihilo de diferentes *suites* de *benchmarks*: Rodinia [15], PARSEC [12], NAS Parallel Benchmarks [11]. Además, se han incluido el benchmark RNASeq [18], un benchmark basado en OpenMP que simula una aplicación de secuencias de ARN, y dos *benchmarks* multihilo: BLAST, una aplicación bioinfórmatica, y FFTW3D, un benchmark que realiza la transformada rápida de Fourier. También hemos experimentado con

PBBCache, un simulador de particionado de caché paralelo desarrollado en Python [20].

5.2.2. CloudSuite

Los *benchmarks* de Cloudsuite están basados en pilas de software del mundo real. Queremos clasificarlos según sus comportamientos, ya que no hay documentación detallada sobre el rendimiento de cada *benchmark*. Los *benchmarks* de Cloudsuite utilizadas en nuestro análisis son:

- **Data Analytics**: Este *benchmark* está diseñado para ejecutarse en un clúster de Hadoop, donde un nodo maestro (*master*) se encarga de ejecutar el programa controlador, mientras que los nodos trabajadores (*workers*) realizan las operaciones de mapeo y reducción. El nodo maestro coordina las tareas distribuidas y asigna trabajos a los nodos trabajadores, que procesan grandes volúmenes de datos en paralelo. Esto permite realizar análisis complejos y procesamiento de datos a gran escala de manera eficiente.

- **Data Caching**: Este *benchmark* utiliza el servidor Memcached [6], ampliamente utilizado por muchos sitios web en la actualidad. Simula el comportamiento de un servidor de Memcached utilizando un dataset de Twitter. Para su ejecución, es necesario lanzar al menos dos contenedores: uno para el servidor Memcached y otro para el cliente que solicita los datos almacenados en el servidor. Esta carga de trabajo requiere una estricta garantía de Calidad de Servicio (QoS): la latencia en el percentil 99 debe ser inferior a 1 ms.

- **Data Serving Relational**: Este *benchmark* está basado en PostgreSQL y tiene dos modos de funcionamiento: TPC-C y OLTP-RW, siendo este último uno de los más utilizados. Para su ejecución, es necesario lanzar dos contenedores, uno para el cliente y otro para el servidor, para garantizar su funcionalidad. Al final de su ejecución, el *benchmark* proporciona estadísticas como TPS (Transacciones Por Segundo), la media del percentil 95, entre otros datos relevantes.

- **Data Serving MySQL**: Como mencionamos en la sección 2.3.1, este benchmark es una creación nuestra basada en el anterior *benchmark* Data Serving Relational. Este *benchmark* es muy similar a Data Serving Relational, pero utiliza MySQL en lugar de PostgreSQL.

- **Graph Analytics**: Este *benchmark* está basado en el framework Spark, un proyecto de código abierto de Apache, y está diseñado para realizar análisis de grafos sobre *datasets* a gran escala. Ofrece tres modos de ejecución correspondientes a los tres algoritmos que puede ejecutar utilizando la API de GraphX: PR (*Page Rank*), CC (*Connected Components*) y TC (*Triangle Count*).

- **In-Memory Analytics**: Este *benchmark* está basado en Apache Spark y simula el procesamiento analítico automatizado para agrupar, clasificar y filtrar

información. Ejecuta un algoritmo de filtrado colaborativo proporcionado por Spark MLlib en memoria, utilizando un *dataset* de calificaciones de usuarios a películas proporcionado por Movielens [2].

5.3. Clases de *benchmarks*

Para poder garantizar el aislamiento entre las distintas aplicaciones que se ejecutan en un mismo servidor, es imprescindible conocer cómo estas se comportan. Además esto nos ayuda a saber cómo se ha de particionar la caché para sacar el máximo provecho a los recursos de la CPU y afectar lo mínimo al rendimiento y/o latencia de las distintas aplicaciones.

Según el comportamiento de las aplicaciones con respecto al uso de estos recursos compartidos y su rendimiento, dependiendo del acceso que tengan a estos, tenemos la clasificación de tres categorías definida en [30]; que constituye las siguientes clases:

- **Light-sharing:** Estos programas son poco intensivos en CPU y no hacen un gran uso de la memoria cache compartida, ya que los datos (*working set*) que estos requieren caben perfectamente en su cache de segundo nivel. Esto implica que su rendimiento no se ve afectado por el número de vías de cache que disponga.
- **Cache-sensitive:** Dentro de esta clase se encuentran los programas, que como bien indica su nombre, su rendimiento se ve seriamente afectado dependiendo del número de vías de cache de tercer nivel dispongan. Estos programas requieren de más espacio en memoria que los Light-sharing y por tanto necesitan del suficiente espacio en la cache L3 para poder tener todos los datos necesarios para su ejecución y así evitar la degradación del rendimiento debido a fallos de acceso a cache.
- **Streaming:** Esta clase al igual que los programas Light-sharing, apenas se ve influenciado su rendimiento por el número de vías de cache que disponen. Sin embargo estos se tratan de programas intensivos en memoria y con alto consumo de ancho de banda. El tamaño del *working set* de esta clase de programas es bastante grande y apenas se reusan datos de este, lo que provoca que sin importar del tamaño de cache L3 que dispongan, se eviten un gran número de fallos de acceso a memoria y en consecuencia una degradación del rendimiento.

La descripción de estas clases revela la importancia de caracterizar los programas que se lanzan, ya que programas de tipo *streaming* pueden degradar gravemente el rendimiento de otras aplicaciones, si estos acaparan a demasiado espacio de la cache de tercer nivel. También se puede dar el caso de que programas de tipo cache-sensitive no tengan a su disposición el suficiente espacio de cache produciendo un aumento en su tiempo de ejecución y latencia.

Mediante el lanzamiento de estas con distintos números de vías asignadas y tras tener los resultados de las distintas ejecuciones, para cada número de vías escogido, se puede hacer un análisis de como estas han afectado a su latencia, consumo de ancho de banda y tiempo de ejecución. Y así conocer a que clase pertenecen estas aplicaciones.

5.4. Fase experimental

Una vez definido el tipo de clasificación a usar, siendo este el que define tres clases de aplicaciones anteriormente explicado. Se plantean una serie de experimentos, con el fin de lograr clasificar los distintos *benchmarks* cloud 5.2.2 y HPC 5.2.1 que disponemos. Los experimentos se han hecho en distintas fases, las cuales son las siguientes:

1. En la primera fase, necesitamos determinar los valores adecuados de R/TPS (*Requests/Transactions Per Second*) para los *benchmarks* cliente-servidor proporcionados por Cloudsuite. Nuestro objetivo es identificar el umbral en el que la latencia del *benchmark* empieza a crecer de forma desorbitada al ejecutarse con un conjunto fijo de núcleos.

2. En la siguiente fase, conociendo ya los valores adecuados de R/TPS, realizamos un análisis del consumo de ancho de banda, número de hilos activos y latencia de los distintos *benchmarks*. Esto con el fin de saber el comportamiento de los distintos programas e ir teniendo una cierta idea de la clase de programa a la cual pueden pertenecer.

3. En la tercera fase, se realiza una caracterización de todas las aplicaciones otorgándoles una cantidad variable de espacio en la LLC. Esto nos permite estudiar la sensibilidad a la caché y la contención, evaluando cómo afectan estas restricciones al rendimiento y comportamiento de cada aplicación.

4. En la cuarta y última fase, realizaremos un estudio de la contención utilizando pares de aplicaciones HPC y Cloud. Ejecutaremos dos experimentos para cada par de aplicaciones: uno sin particionado y otro con particionado de cache. Esto nos permitirá evaluar el impacto del particionado en la contención y el rendimiento de las aplicaciones.

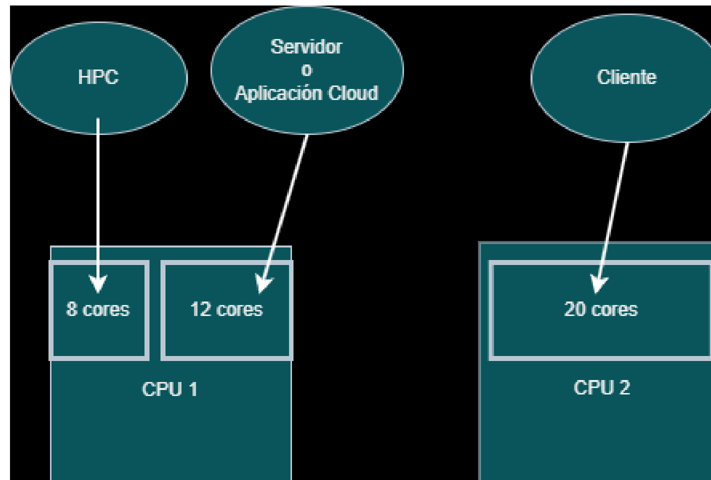


Figura 5.3: Esquema distribución de los benchmarks

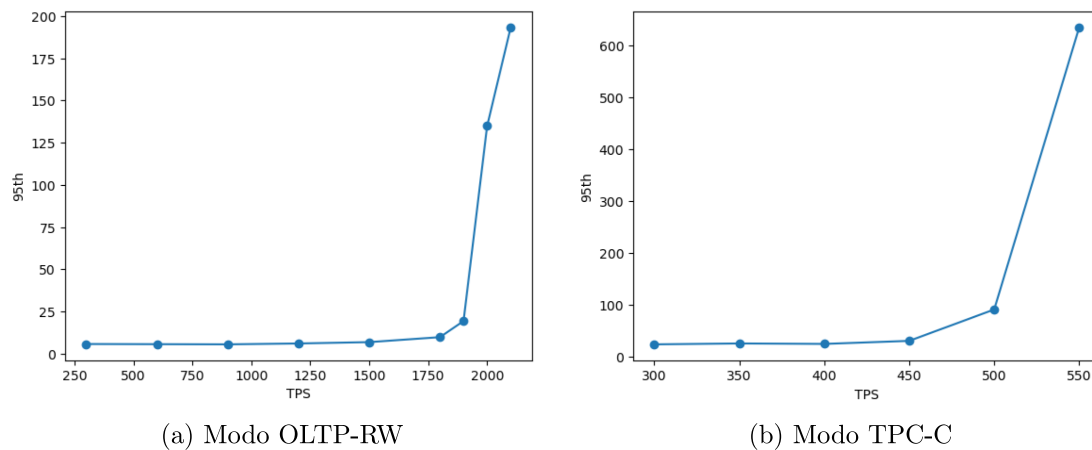
Para estos experimentos se hizo uso de la Plataforma Skylake (plataforma descrita en la sección 5.1.2). La figura 5.3 muestra un esquema de la distribución de los recursos de la plataforma Skylake para los distintos *benchmarks*. Se definió el uso de 12 cores por parte de las aplicaciones cloud y 8 por parte de los *benchmarks* HPC. Se otorga de 8 cores a los programas HPC debido a que muchos de ellos trabajan con potencias de dos número de cores. Esta nos parecía la potencia de dos más adecuada, a que es lo suficientemente grande para no afectar al rendimiento de los programas HPC, sin ser demasiado grande como para dejar a los *benchmarks* cloud con demasiados pocos cores. Por último, cabe destacar el uso de las 2 CPUs con las que cuenta esta plataforma para el lanzamiento de los *benchmarks* cloud cliente-servidor. Otorgando los 12 cores para *benchmarks* cloud a la aplicación servidor y otorgando a la aplicación cliente (generador de cargas del servidor) el uso de los 20 cores de la otra CPU para que así este pueda estresar al servidor, sin interferir con la jerarquía de memoria de las primeras CPU.

5.4.1. Barridos de R/TPS para *benchmarks* de CloudSuite

Para identificar el umbral en el que se alcanza el punto se saturación del *benchmark* (cuando la latencia del *benchmark* empieza a crecer de forma desorbitada), hemos realizado barridos de R/TPS (*Requests/Transactions Per Second*). Luego, seleccionamos un punto ligeramente por debajo de este umbral para evitar la saturación. Este experimento se ha llevado a cabo en los siguientes *benchmarks* de tipo cliente-servidor de Cloudsuite: Data Caching, Data Serving Relational (modos TPC-C y OLTP-RW), y Data Serving MySQL (modos TPC-C y OLTP-RW).

Los barridos de R/TPS se realizaron utilizando la herramienta Het-Harness y su comando *harness-docker* (descrito en el capítulo **??**). En cada ejecución del *benchmark*, se generó un archivo log que contiene todas las salidas del *benchmark*, incluida la información de latencia que nos interesaba, como el percentil 99 para Data Caching y el percentil 95 para el resto de los *benchmarks*. Es importante destacar que

Data Caching requiere una estricta calidad de servicio, asegurando que el percentil 99 de la latencia sea inferior a 1 ms.

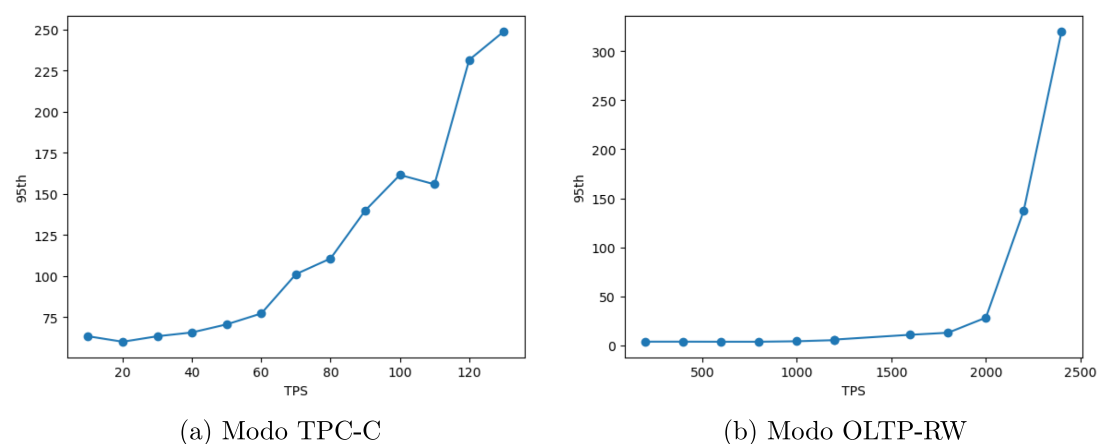


(a) Modo OLTP-RW (b) Modo TPC-C

Figura 5.4: Gráficas de latencia de Data Serving Relational según el TPS

En las gráficas de latencia de Data Serving Relational, representadas en la figura 5.4, se muestran dos gráficos correspondientes a los dos modos operativos del *benchmark*. Observamos que a medida que aumenta el TPS (*Transaction Per Second*), el percentil de latencia también aumenta. Sin embargo, llegado a un cierto número de TPS, la latencia experimenta un incremento significativo. El objetivo es identificar un punto de TPS antes de que el percentil de latencia se dispare, manteniendo un margen de seguridad para evitar la saturación del sistema.

Tras analizar las gráficas, hemos seleccionado 400 TPS para el modo TPC-C, donde la latencia se mantiene por debajo de 30 ms, y 1600 TPS para el modo OLTP-RW, con una latencia inferior a 10 ms.



(a) Modo TPC-C (b) Modo OLTP-RW

Figura 5.5: Gráficas de latencia de Data Serving MySQL según el TPS

En la figura 5.5 se muestran las dos gráficas de latencia del *benchmark* Data Serving MySQL, que representan los dos modos de operación. Después de analizar

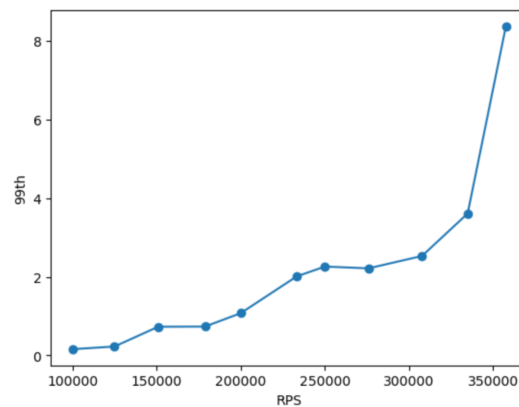


Figura 5.6: Gráfica de latencia de Data Caching según el TPS

estas gráficas, hemos seleccionado 50 TPS para el modo TPC-C, donde la latencia se mantiene por debajo de 100 ms, y 1100 TPS para el modo OLTP-RW, que no supera los 6 ms de latencia.

En la figura 5.6, se presenta la gráfica de latencia del *benchmark* de Data Caching. Observamos que utiliza el percentil 99th como el eje Y, ya que este *benchmark* requiere una calidad de servicio más estricta que los anteriores. Específicamente, exige que la latencia del percentil 99 no supere los 1 ms. Por este motivo, hemos seleccionado 150k RPS, ya que mantiene la latencia por debajo de 1 ms.

Aplicaciones	**Latencia (ms)**	**R/TPS escogido**
PostgreSQL (OLTP-RW)	<10ms	1600 TPS
PostgreSQL (TPC-C)	24ms	400 TPS
MySQL (OLTP-RW)	5ms	1100 TPS
MySQL (TPC-C)	<100ms	50 TPS
Memcached	<1ms	150k RPS

Tabla 5.1: Aplicaciones con las medidas escogidas y su latencia

Viendo la tabla 5.1, se presentan los valores de R/TPS seleccionados para cada *benchmark* junto con su respectiva latencia en milisegundos. Esta tabla resume las decisiones tomadas durante la fase de determinación de los R/TPS adecuados para cada aplicación, teniendo en cuenta sus requisitos de latencia crítica.

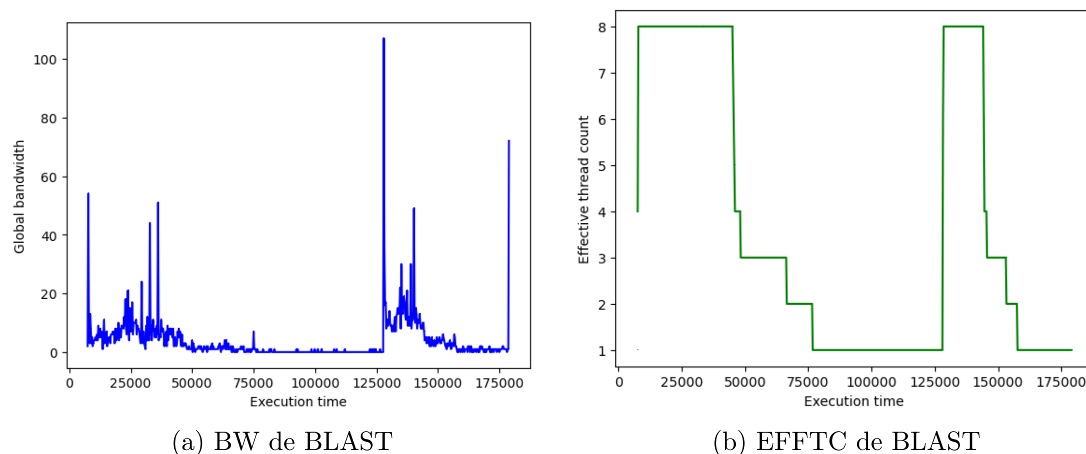
Los experimentos de barrido de R/TPS, así como los próximos que presentaremos, se han llevado a cabo en la plataforma experimental final descrita en la sección 5.1.2. En esta plataforma es donde hemos realizado la mayoría de los experimentos.

5.4.2. Experimentos de caracterización I: Latencia/EFFTC/BW

En esta fase de los experimentos, hemos ejecutado todas las aplicaciones de forma individual, tanto las pertenecientes a Cloudsuite como las de HPC, con el objetivo de obtener información sobre su comportamiento en términos de Ancho de Banda (BW, *Bandwidth*), Hilos Activos (EFFTC, *Effective Thread Count*, que es el número de hilos activos dentro del contenedor) y, en el caso de las aplicaciones cliente-servidor, su latencia. Los resultados obtenidos proporcionan una primera impresión sobre las características de estas aplicaciones. Por ejemplo, podemos determinar si una aplicación es del tipo Streaming (si consume una gran cantidad de BW) o Light Sharing (si su uso de BW es mínimo), así como analizar la relación entre los picos de BW y la cantidad de hilos activos en un momento dado. Además, podemos estudiar el comportamiento de la latencia mediante el análisis de las gráficas de BW y EFFTC. Se ha utilizado un *plugin custom* que se ha construido para medir el BW (proporcionado por la API de PMCTrack) y el número de hilos activos utilizando una técnica descrita en el artículo [29].

Este primer experimento de caracterización nos proporciona una aproximación inicial sobre el tipo de aplicaciones que estamos ejecutando. Sin embargo, es importante tener en cuenta que únicamente con el BW, el EEFTC y la latencia, no podemos deducir exactamente el tipo y comportamiento preciso de estas aplicaciones.

A continuación se muestran los resultados y el análisis de estos, para los *benchmarks* de cloud y aquellos *benchmarks* HPC de los cuales no se tenía registro de su clase. Además de los resultados de los experimentos de algunas aplicaciones, los cuales han resultado interesantes para el uso de estos programas en otros experimentos posteriores.



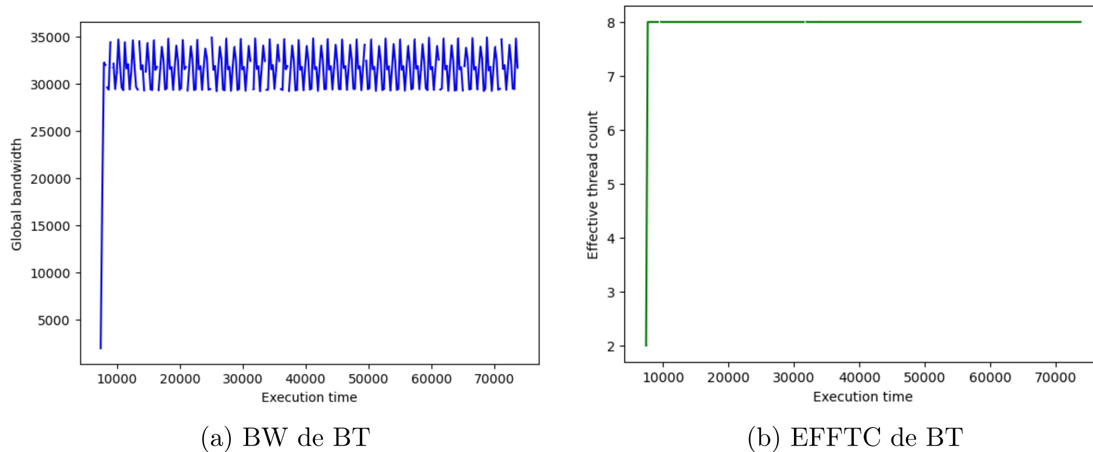
(a) BW de BLAST

(b) EFFTC de BLAST

Figura 5.7: Gráficas de BW/EFFTC de BLAST

En la figura 5.7, que muestra las gráficas de BW y EFFTC de la aplicación BLAST, podemos observar que esta aplicación se identifica como un caso de Light-

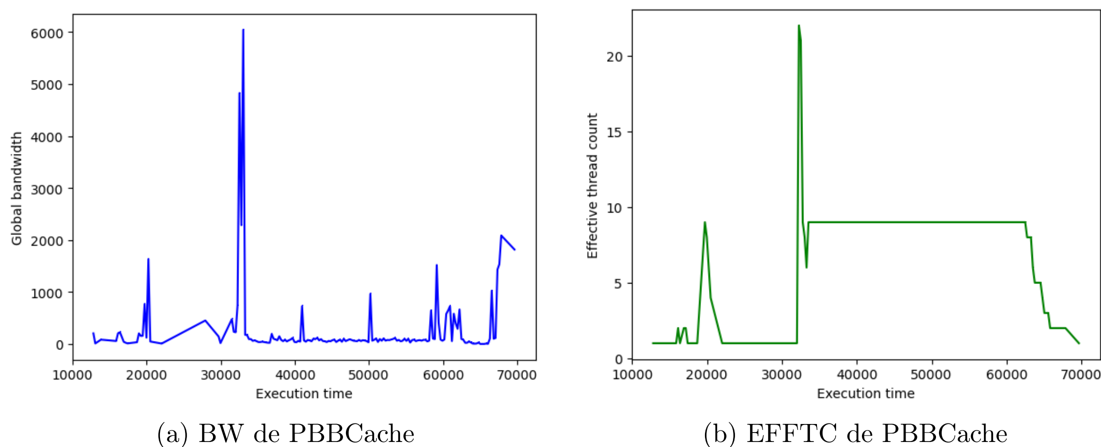
sharing. Aunque inicialmente utiliza 8 hilos activos, su ancho de banda se mantiene por debajo de 100 MB/s y, en general, se estabiliza en menos de 40 MB/s. La cantidad de hilos activos aumenta al principio, coincidiendo con el aumento del ancho de banda, pero luego disminuye a solo 1 hilo activo.



(a) BW de BT (b) EFFTC de BT

Figura 5.8: Gráficas de BW/EFFTC de BT

En la figura 5.8 podemos observar las gráficas de BW y EFFTC de la aplicación BT. De esta aplicación, ya tenemos registro de su clasificación, siendo clasificada como cache-sensitive [13]. Sin embargo, el alto consumo de ancho de banda observado,aproximadamente unos 32 GB/s y su uso de 8 hilos constante, nos hace pensar que esta pueda tratarse más bien de una aplicación streaming.

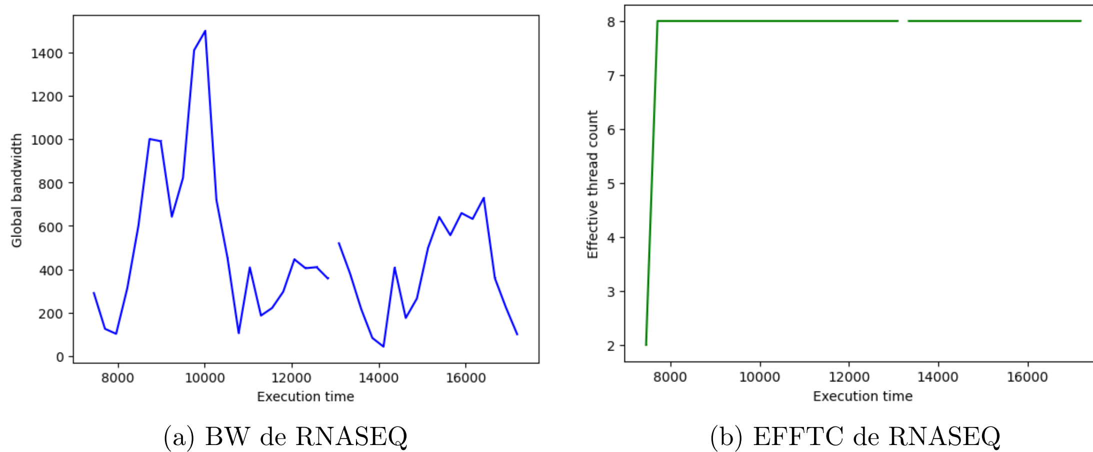


(a) BW de PBBCache (b) EFFTC de PBBCache

Figura 5.9: Gráficas de BW/EFFTC de PBBCache

Como observamos en la figura 5.9, la cual describe el uso de ancho de banda y hilos activos del *benchmark* PBBCache. Esta se presenta como aplicación intensiva en CPU debido a la cantidad de hilos activos durante la ejecución, siendo estos la mayor parte de la ejecución 10 hilos activos y llegando a picos de 20. Con respecto al ancho de banda, esta apenas consume. Hasta teniendo en cuenta el número de

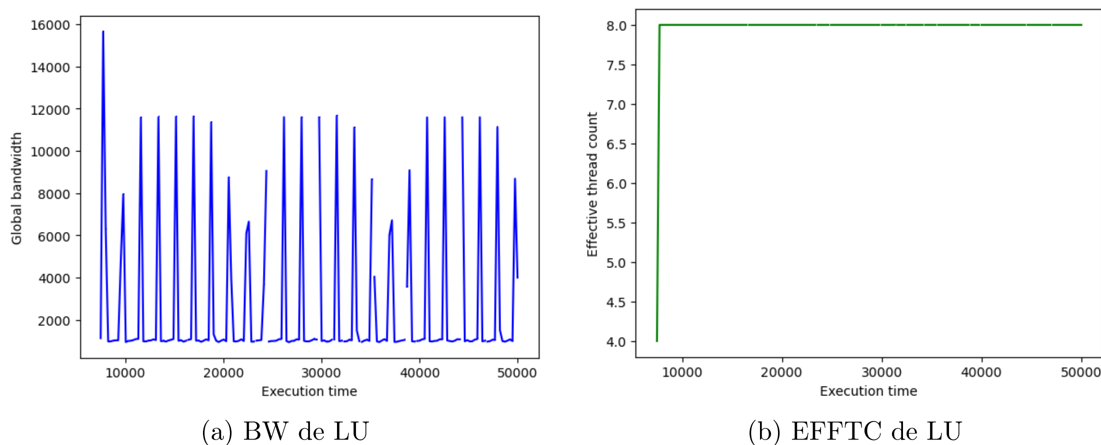
hilos activos, encontramos un pico en el consumo de ancho de banda que coincide con el pico que produce la activación de los 20 hilos. No obstante, por lo general este programa presenta un bajo consumo de ancho de banda.



(a) BW de RNASEQ (b) EFFTC de RNASEQ

Figura 5.10: Gráficas de BW/EFFTC de RNASEQ

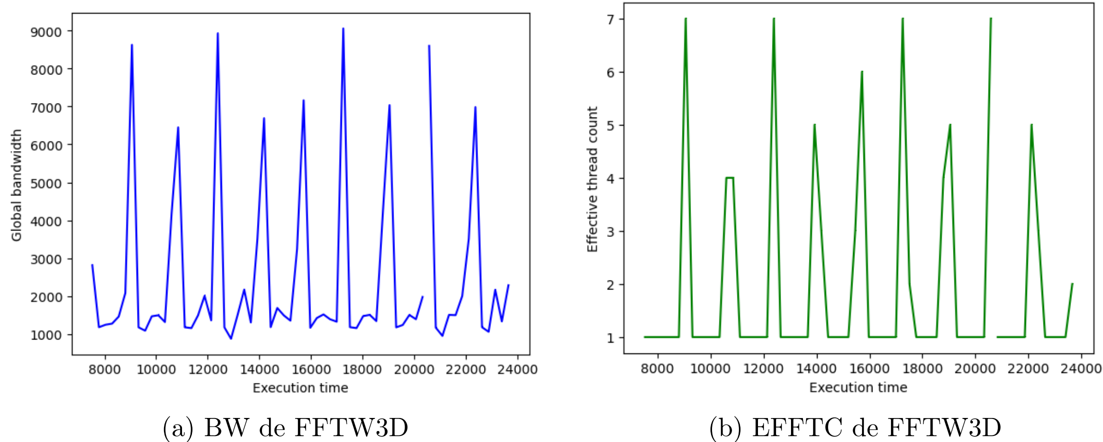
En la figura 5.10, la cual muestra el consumo de ancho de banda y número de hilos activos del *benchmark* RNASEQ, podemos observar como esta por lo general presenta un *bandwidth* moderado salvo por un pico de 1.4 GB/s. Este exhibe un comportamiento muy intenso en CPU, teniendo 8 hilos activos prácticamente durante toda su ejecución. Puede que se trate de un programa cache-sensitive debido al pico en *bandwidth*, pero estos datos no son suficientes para su clasificación.



(a) BW de LU (b) EFFTC de LU

Figura 5.11: Gráficas de BW/EFFTC de LU

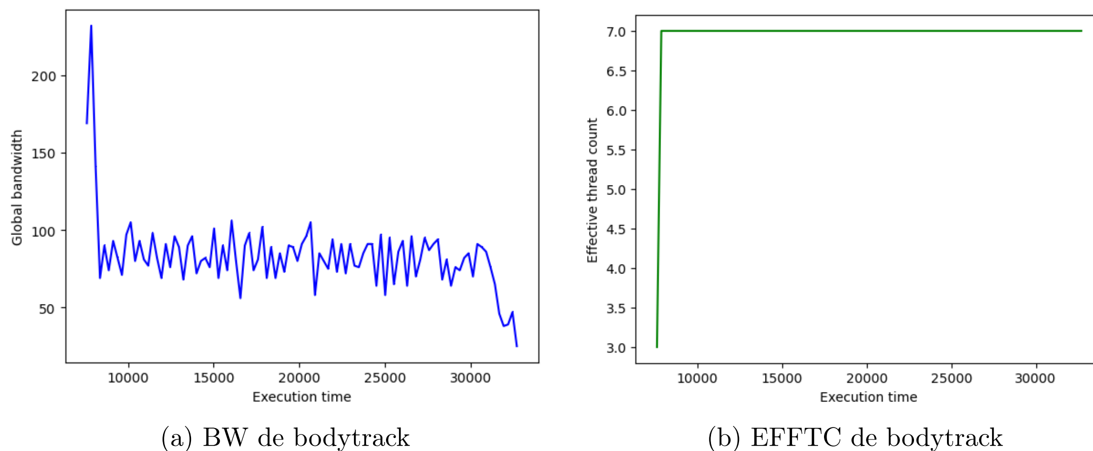
Como se observa en la figura 5.11, la cual describe el consumo de *bandwidth* e hilos activos del programa HPC LU, este presenta un consumo de ancho de banda considerable, sin llegar a ser demasiado grande, teniendo una media de unos 8 GB/s. Presenta un alto uso de CPU, el cual se mantiene a lo largo de toda su ejecución, teniendo siempre 8 hilos activos. De esta aplicación se tiene registro de su clase [13], sin embargo sólo con estos datos no es posible confirmar la clase de esta.



(a) BW de FFTW3D (b) EFFTC de FFTW3D

Figura 5.12: Gráficas de BW/EFFTC de FFTW3D

En la figura 5.12, que describe el uso de bandwidth y el númeor de hilos activos para el *benchmark* FFTW3D, podemos observar com este tiene varios picos de forma regular tanto en ancho de banda como en hilos activos. Estos picos coinciden entre las dos gráficas, lo que explica las repentinas subidas de consumo de ancho de banda, debido al aumento de número de hilos activos. Este en sus pico, presenta consumos de ancho de banda algo elevados y bastantes hilos activos, podría tratarse de una aplicación streaming o cache-sensitive, pero estos datos no son suficientes para su caracterización.

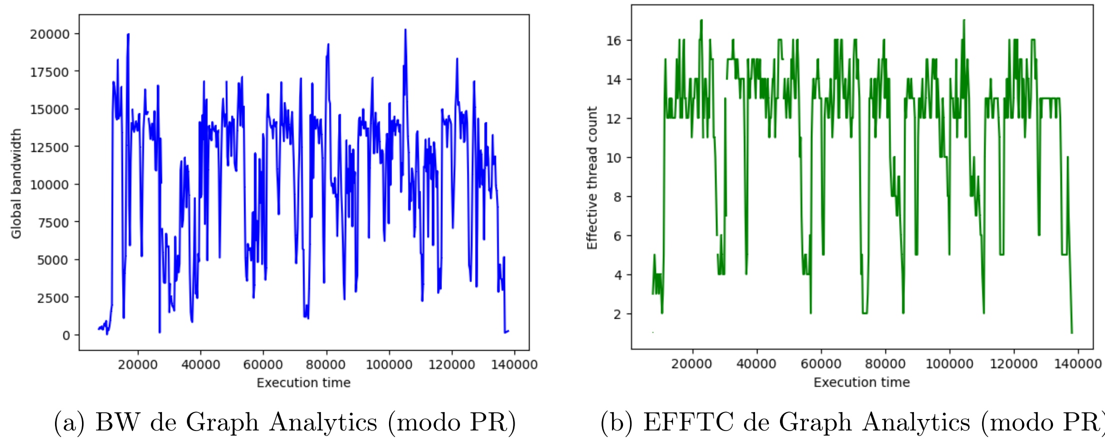


(a) BW de bodytrack (b) EFFTC de bodytrack

Figura 5.13: Gráficas de BW/EFFTC de bodytrack

Observando la figura 5.13, la cual describe el consumo de ancho de banda y número de hilos activos para el programa *bodytrack*, vemos que este apenas consume un ancho de banda sustancial a lo largo de su ejecución. Este presenta una gran cantidad de hilos activos, teniendo 7 hilos activos a lo largo de su ejecución, esto lo puede llegar a hacer algo intensivo en CPU. Sin embargo, su consumo de ancho de banda, es muy bajo a pesar de los hilos activos, estando este muy por debajo de 300 MB/s. De este *benchmark* se tiene registro de su clase, siendo esta cache-sensitive [13]. Sin embargo su bajo consumo de *bandwidth* nos hace dudar de esta

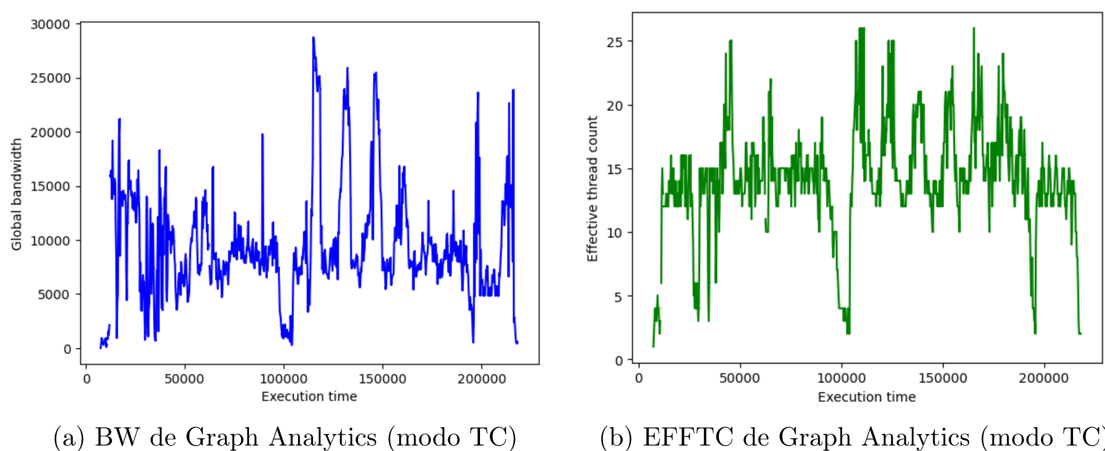
clasificación, pudiendo ser este light-sharing. Por el momento no podemos clasificar de forma definitiva el programa.



(a) BW de Graph Analytics (modo PR) (b) EFFTC de Graph Analytics (modo PR)

Figura 5.14: Gráficas de BW/EFFTC de Graph Analytics modo PR

En la figura 5.14, se observan las gráficas de BW y EFFTC de Graph Analytics en el modo PR. Las gráficas muestran un patrón de BW y EFFTC muy irregular e inestable: el BW varía desde unos 2.5 GB/s hasta picos de 17.5 GB/s, mientras que el EFFTC coincide con estas fluctuaciones, alcanzando hasta 16 hilos activos en los picos. Estos resultados indican que la aplicación es muy intensiva en CPU. Aunque viendo el BW podríamos clasificarlo como una aplicación de tipo streaming, pero será necesario observar su comportamiento en experimentos adicionales para clasificarla de manera más precisa.

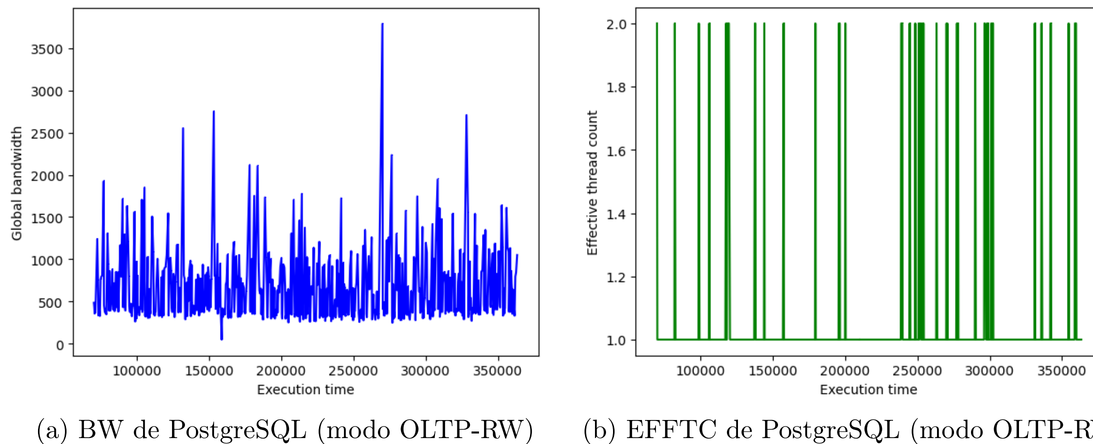


(a) BW de Graph Analytics (modo TC) (b) EFFTC de Graph Analytics (modo TC)

Figura 5.15: Gráficas de BW/EFFTC de Graph Analytics modo TC

En la figura 5.15, se muestran las gráficas de BW y EFFTC de Graph Analytics en el modo TC. Al igual que en el modo PR, estas gráficas también muestran una irregularidad en el BW y EFFTC. Los picos de EFFTC coinciden con los picos de BW, y en este modo, la aplicación es aún más intensiva en CPU, llegando a utilizar más de 20 hilos activos durante su ejecución. En la gráfica de BW, se observa que

varía entre aproximadamente 5 GB/s y 20 GB/s, lo que sugiere que también podría clasificarse como una aplicación de tipo streaming. Sin embargo, para clasificarla con mayor precisión, será necesario evaluar si reducir los hilos activos afecta el rendimiento de la aplicación, lo que podría indicar que se trata de una aplicación cache-sensitive.



(a) BW de PostgreSQL (modo OLTP-RW) (b) EFFTC de PostgreSQL (modo OLTP-RW)

Figura 5.16: Gráficas de BW/EFFTC de Data Serving Relational modo OLTP-RW

Como se puede apreciar en la figura 5.16, que muestra las gráficas BW y EFFTC de Data Serving Relational en el modo OLTP-RW, la aplicación presenta un bajo BW, con una media alrededor de 1 GB/s. Además, tiene un consumo de CPU muy reducido, con un máximo de dos hilos activos durante su ejecución.

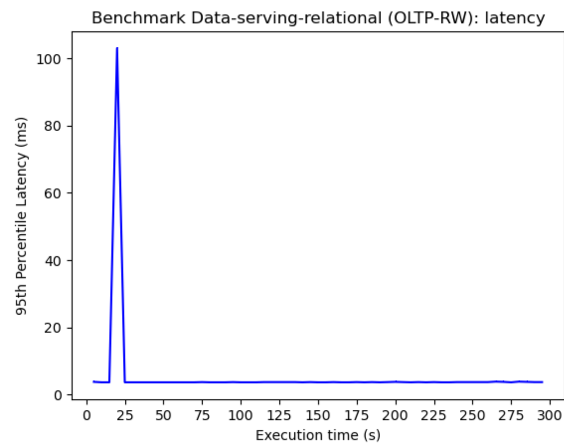


Figura 5.17: Gráficas de Latencia de Data Serving Relational modo OLTP-RW

En la figura 5.17, se muestra la gráfica de latencia de la aplicación Data Serving Relational en el modo OLTP-RW. Se observa que, en general, la latencia se mantiene relativamente estable a lo largo de la ejecución, con algunos picos notables, siendo el más significativo el que alcanza los 100 ms. Este pico se produce al inicio de la ejecución, debido a la carga inicial de datos en la caché y la preparación del entorno para la ejecución del *benchmark* (fase de *warmup*).

Como podemos observar, algunas aplicaciones presentan características confusas, haciendo que solamente con el BW y EFFTC no sea posible clasificarlas de forma clara. Esto subraya la necesidad de realizar más experimentos para obtener una clasificación más precisa y completa. Es fundamental determinar si estas aplicaciones son del tipo streaming o sache-sensitive, o si requerirán una categorización adicional para comprender mejor su comportamiento.

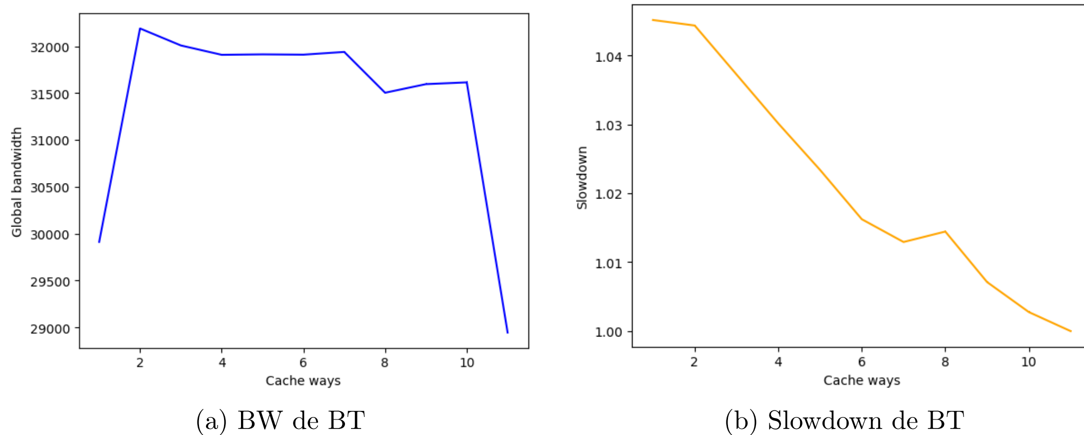
5.4.3. Experimentos de caracterización II

Con los resultados obtenidos en la fase anterior, tras medir el consumo de ancho de banda, número de hilos activos y en algunos casos latencia, se pudo hacer una idea del comportamiento de algunos de los programas hasta llegar a tener una idea bastante clara de su clase. Sin embargo este análisis no resultó suficiente para la caracterización de muchos otros programas. Es por esto que con el fin de lograr caracterizar el conjunto de *benchmarks*, se procede a estudiar cómo afecta el grado de ocupación de la LLC que disponen los *benchmarks* a su rendimiento.

5.4.3.1. Barridos de vías de caché

Para lograr estudiar el efecto de la ocupación de la LLC en el rendimiento de cada uno de los *benchmarks*, se procede a realizar un experimento de barrido de vías de caché. Cuando nos referimos a un experimento de barrido de vías, nos referimos a la ejecución de un programa variando el número de vías de caché disponibles, comenzando desde el mínimo número de vías hasta el máximo. Para cada número de vías de caché dentro de este rango, se realiza una ejecución del programa. Este tipo de experimento requiere una CPU que permita el particionado de caché por vías, como es el caso de nuestra plataforma Skylake. En nuestro experimento, con una caché de 11 vías disponibles, el barrido de vías se realiza desde 1 hasta 11 vías.

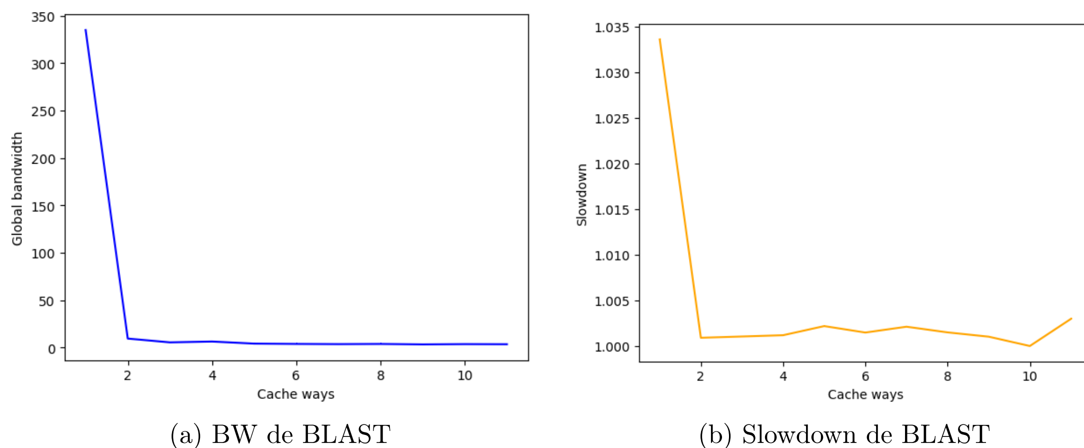
Para analizar el efecto que tiene el espacio usado de la LCC en los distintos programas, se comparará la media del consumo de ancho de banda, el *slowdown* (degradación del rendimiento, que se verá reflejado en el incremento del tiempo de ejecución) y la media de la latencia según el número de vías para los *benchmarks* de cloud cliente-servidor. En este análisis, hemos definido una bajada del *slowdown* del 0.1 como umbral para concluir una mejora o empeoramiento del rendimiento considerable en el tiempo de ejecución según el número de vías.



(a) BW de BT (b) Slowdown de BT

Figura 5.18: Gráficas de BW y Slowdown de BT

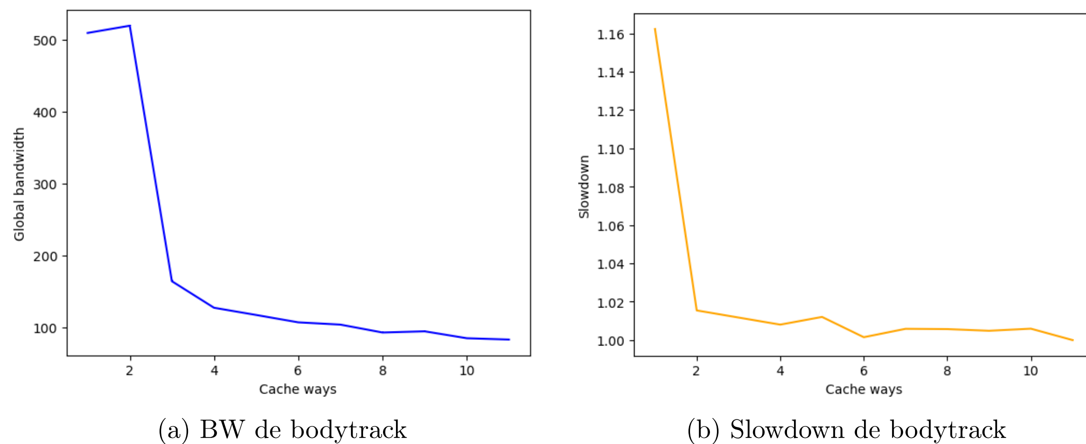
La figura 5.18 nos muestra cómo afecta el número de vías de caché del que dispone BT a su consumo de ancho de banda y a su tiempo de ejecución (medido en slowdown). Podemos observar cómo este presenta un consumo de ancho de banda muy elevado, el cual apenas baja en 11 vías de los 30 GB/s. Donde si parece afectar el número de vías es al observar su slowdown. Sin embargo esta bajada no llega a ser del 0.1 y por tanto no consideramos que el número de vías afecte a su tiempo de ejecución. Con estos datos, podemos considerar esta aplicación como una del tipo streaming.



(a) BW de BLAST (b) Slowdown de BLAST

Figura 5.19: Gráficas de BW y Slowdown de BLAST

En la figura 5.19 se muestran las gráficas de la media de BW y del slowdown en función de las vías de caché disponibles. Podemos observar que esta aplicación tiene un consumo de ancho de banda muy bajo y que, al asignarle 2 vías de caché, dicho consumo disminuye considerablemente. Con una vía de caché, el consumo puede superar los 300 MB/s, pero al asignar 2 vías, este consumo se reduce a menos de 50 MB/s. El slowdown es mínimo; aunque baja ligeramente con 2 vías de caché, no alcanza ni siquiera 0.1. Esto es un claro ejemplo de una aplicación tipo light-sharing.

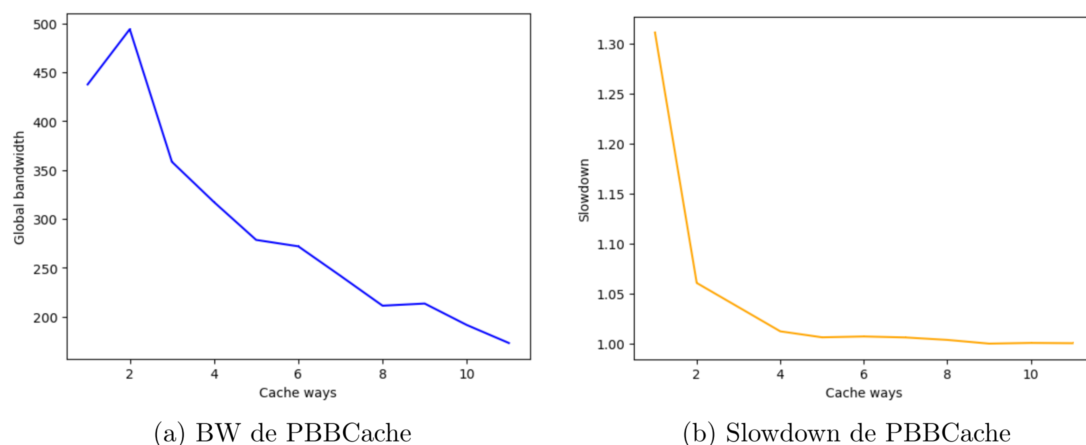


(a) BW de bodytrack

(b) Slowdown de bodytrack

Figura 5.20: Gráficas de BW y Slowdown de bodytrack

En la figura 5.20 podemos observar las gráficas de la media de BW y del slowdown de bodytrack según las vías de cache que dispone. Vemos que es muy similar a BLAST, con un bajo uso de BW que disminuye considerablemente con 3 vías de caché: desde 500 MB/s cuando utiliza 1-2 vías, hasta aproximadamente 100 MB/s con 3-4 vías. En cuanto al slowdown, bodytrack muestra una mejor bajada en comparación con Blast, con una disminución a 0.14 al utilizar 2 vías de caché que luego se va manteniendo.

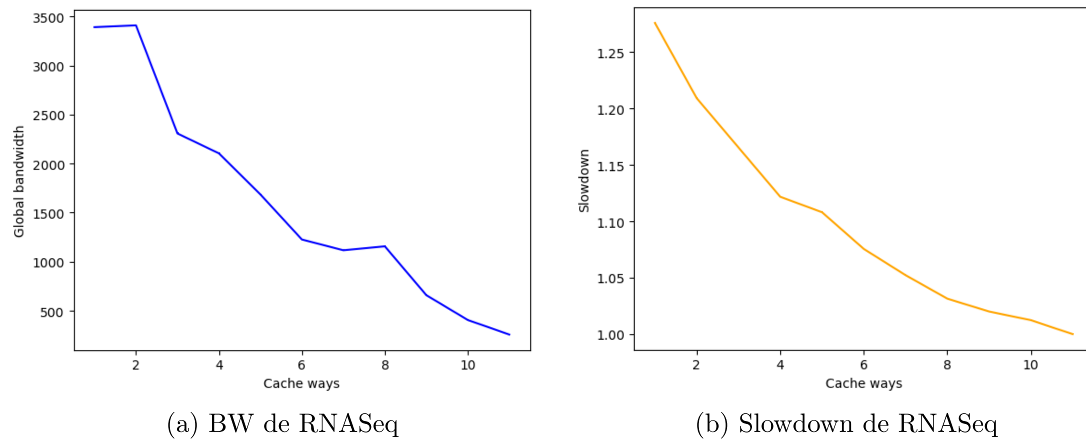


(a) BW de PBBCache

(b) Slowdown de PBBCache

Figura 5.21: Gráficas de BW y Slowdown de PBBCache

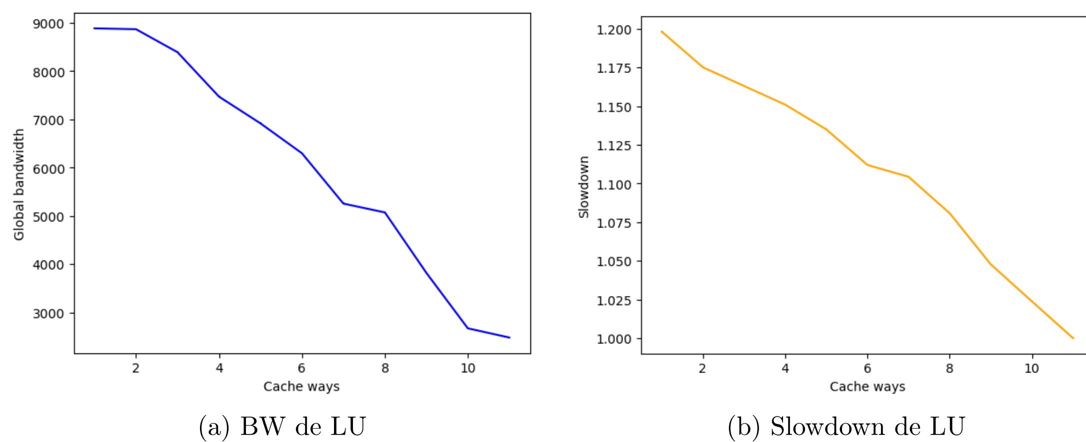
En la figura 5.21 se muestran las gráficas de la media de BW y su tiempo de ejecución en función de las vías de caché disponibles para la aplicación PBBCache. Esta aplicación no hace un uso intensivo de BW durante su ejecución. Sin embargo, no se observa una disminución drástica del BW: utilizando 1-2 vías alcanza hasta 450+ MB/s, y con 3-4 vías baja a aproximadamente 300 MB/s, alcanzando su mínimo con 10-11 vías, donde utiliza menos de 200 MB/s de BW. En cuanto al slowdown, se observa una reducción considerable con 2 vías de caché, bajando de 1.3 a 1.05, lo cual es una mejora notable. Esta aplicación se clasifica como cache-sensitive.



(a) BW de RNASeq (b) Slowdown de RNASeq

Figura 5.22: Gráficas de BW y Slowdown de RNASeq

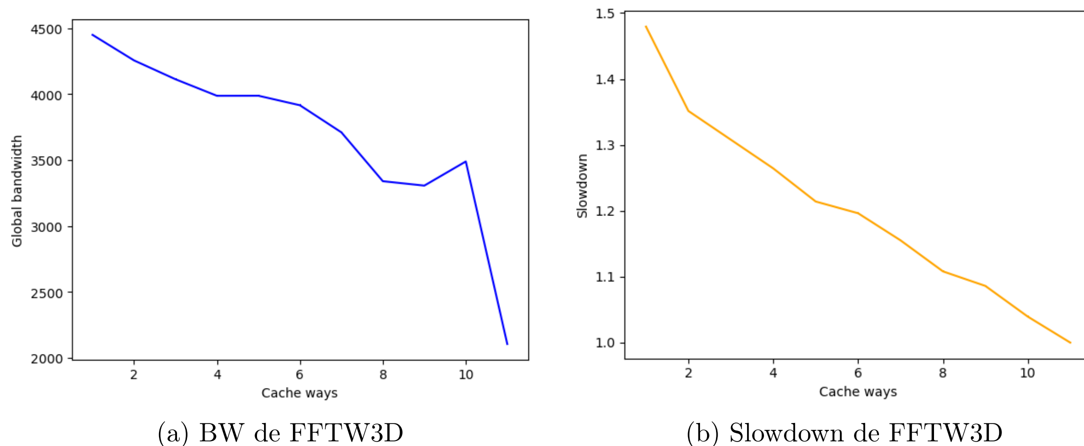
En la figura 5.22 se observan las gráficas de la media de BW y su tiempo de ejecución según las vías de caché disponibles para la aplicación RNASeq. Observamos que el ancho de banda disminuye notablemente a medida que se incrementan las vías de caché disponibles. Con 8 vías de caché, el BW se reduce a 1 GB, en comparación con los 3.5 GB que utiliza cuando dispone de 1-2 vías de caché, lo cual es una reducción significativa. En cuanto al slowdown, se observa una disminución de 1.25 a 1.05 con 8 vías de caché. Podemos clasificar esta aplicación como Cache Sensitive, ya que requiere 8 vías de caché para tener un slowdown reducido.



(a) BW de LU (b) Slowdown de LU

Figura 5.23: Gráficas de BW y Slowdown de LU

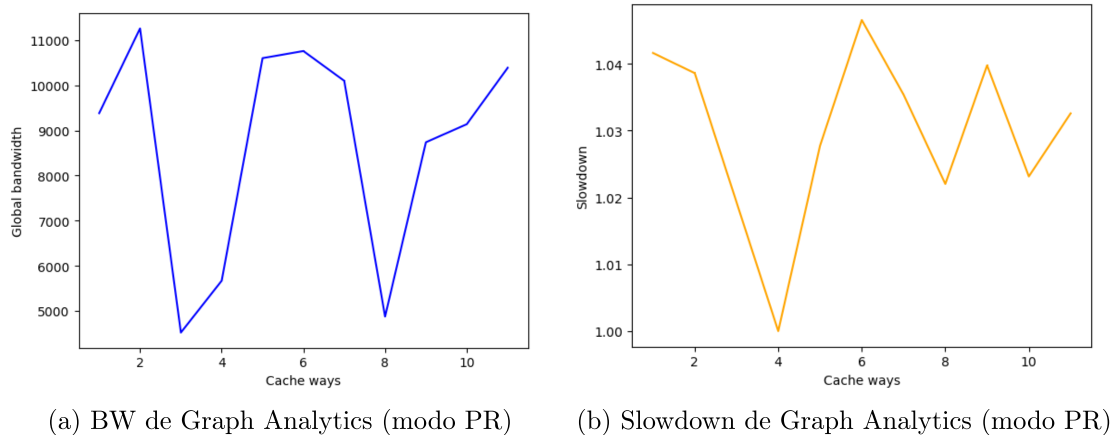
En la figura 5.23 se muestran las gráficas de la media de BW y su tiempo de ejecución según las vías de caché disponibles para la aplicación LU. Esta aplicación hace un uso considerable del BW, alcanzando un pico de aproximadamente 9 GB/s con 1-2 vías de caché. A medida que se incrementan las vías de caché disponibles, el uso de ancho de banda disminuye gradualmente, alcanzando su mínimo con 10-11 vías, donde se reduce a menos de 3 GB/s. En cuanto al slowdown, se observa una disminución de 1.2 a 1.025 con 10-11 vías de caché. Esta aplicación es claramente cache-sensitive y requiere un número significativo de vías de caché para mejorar su rendimiento.



(a) BW de FFTW3D (b) Slowdown de FFTW3D

Figura 5.24: Gráficas de BW y Slowdown de FFTW3D

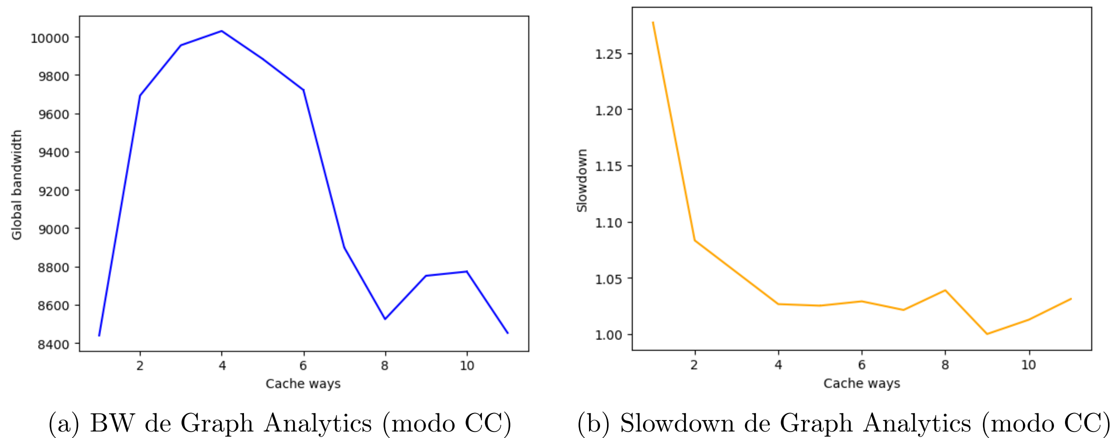
En la figura 5.24 se pueden observar las gráficas de la media de BW y su tiempo de ejecución de la aplicación FFTW3D. Observamos que el BW tiene una tendencia a la bajada, a medida que se incrementan las vías de caché, aunque de forma gradual. Con 1-9 vías de caché, el BW varía entre 3.5 GB/s y 4.5 GB/s, alcanzando su punto más bajo con 11 vías de caché, donde disminuye a 2 GB/s. En cuanto al slowdown, se observa una reducción significativa, bajando de 1.5 a 1 con 11 vías de caché. Considerando todos estos aspectos, esta es una aplicación cache-sensitive, pero similar a LU, requiere un número elevado de vías de caché para lograr un rendimiento adecuado.



(a) BW de Graph Analytics (modo PR) (b) Slowdown de Graph Analytics (modo PR)

Figura 5.25: Gráficas de BW y Slowdown de Graph Analytics (modo PR)

En la figura 5.25, la cual describe la variación del consumo de bandwidth y del tiempo de ejecución según las vías de cache de Graph Analytics (modo PR). Observamos como este presenta un comportamiento muy irregular, esto nos lleva a la conclusión de que este apenas se ve afectado por el número de vías de cache que dispone. El consumo de ancho de banda tiene bajadas considerables para ciertas vías, sin embargo este parece no depender del número de vías de este y el slowdown se mantiene bastante, nunca alcanzando una bajada del 0.1. Por todo esto, este *benchmark*, se trata de un streaming, ya que sus consumos de ancho de banda son muy elevados y a este no le afectan sustancialmente el número de vías de cache asignadas.

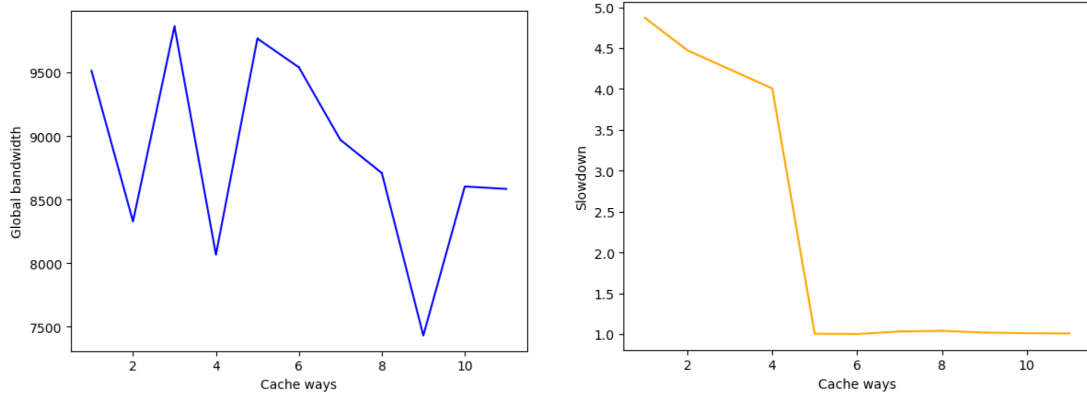


(a) BW de Graph Analytics (modo CC) (b) Slowdown de Graph Analytics (modo CC)

Figura 5.26: Gráficas de BW y Slowdown de Graph Analytics (modo CC)

La figura 5.26 nos muestra como afecta la cache de la que dispone la aplicación Graph Analytics (modo CC) a su tiempo de ejecución y ancho de banda. En estos resultados podemos ver como el consumo de bandwidth es bastante elevado para todo número de vías, pero presentando una bajada de 1,5 GB/s para 8 vías. Con respecto al slowdown, se podría decir que a éste le afecta el número de vías, presentando una bajada de 0,20 entre 2 y 4 vías. Debido al slowdown, se podría decir

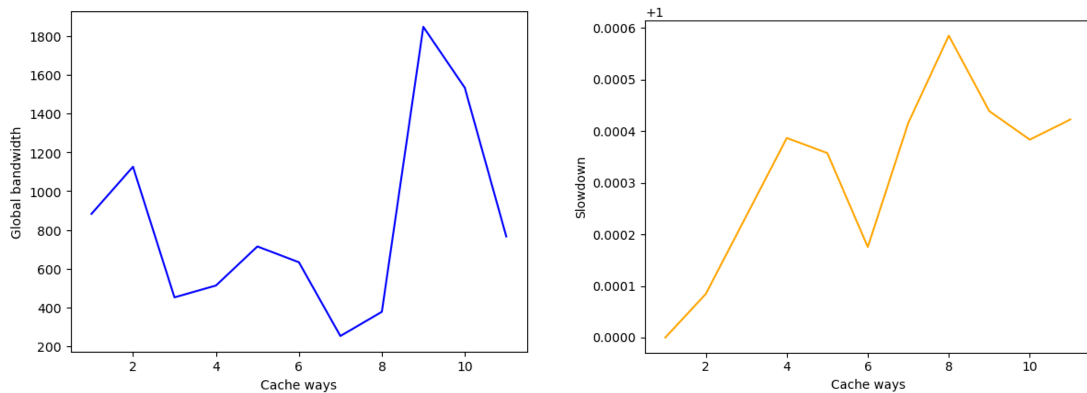
que estamos ante un programa cache-sensitive, pero sin embargo, su alto consumo de ancho de banda, sumado al estancamiento de su slowdown a parti de 2 o 4 vías, nos hace pensar que más bien estamos ante un programa de tipo streaming.



(a) BW de Graph Analytics (modo TC) (b) Slowdown de Graph Analytics (modo TC)

Figura 5.27: Gráficas de BW y Slowdown de Graph Analytics (modo TC)

La figura 5.27 recoge los datos de ancho de banda y slowdown según el número de vías para Graph Analytics (modo TC). En este experimento, podemos observar como apenas se ve afectado el consumo de ancho de banda por el número de vías, presentando bajadas regularmente las cuales no parecen tener relación con el número de vías, si no más bien con el comportamiento del programa. Con respecto al slowdown, podemos observar una bajada enorme de este al disponer de 5 vías, esto nos lleva a clasificar esta aplicación como cache-sensitive, a pesar de su alto consumo de bandwidth, ya que el tiempo de ejecución de este programa está fuertemente influenciado por el número de vías.



(a) BW de Data Serving Relational (modo OLTP-RW) (b) Slowdown de Data Serving Relational (modo OLTP-RW)

Figura 5.28: Gráficas de BW y Slowdown de Data Serving Relational (modo OLTP-RW)

En la figura 5.28 se muestran las gráficas de la media de BW y del tiempo de ejecución en función de las vías de caché disponibles para la aplicación Data

Serving Relational en modo OLTP-RW. Las gráficas indican que el aumento de vías de caché no afecta significativamente su ejecución. El uso de ancho de banda es mínimo, con un pico de 1.8 GB/s cuando se utilizan 9 vías de caché (y en las demás vías se mantiene por debajo de 1 GB/s), y el slowdown es tan reducido que puede considerarse insignificante. Claramente, esta es una aplicación de tipo light-sharing.

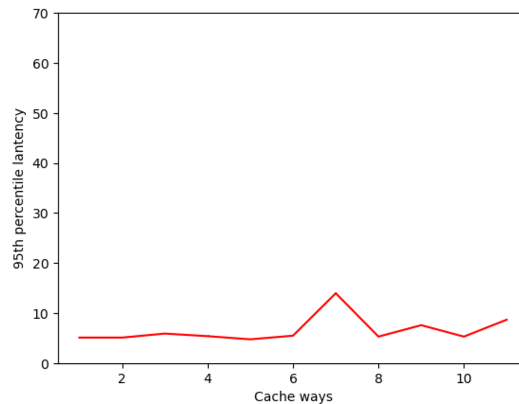
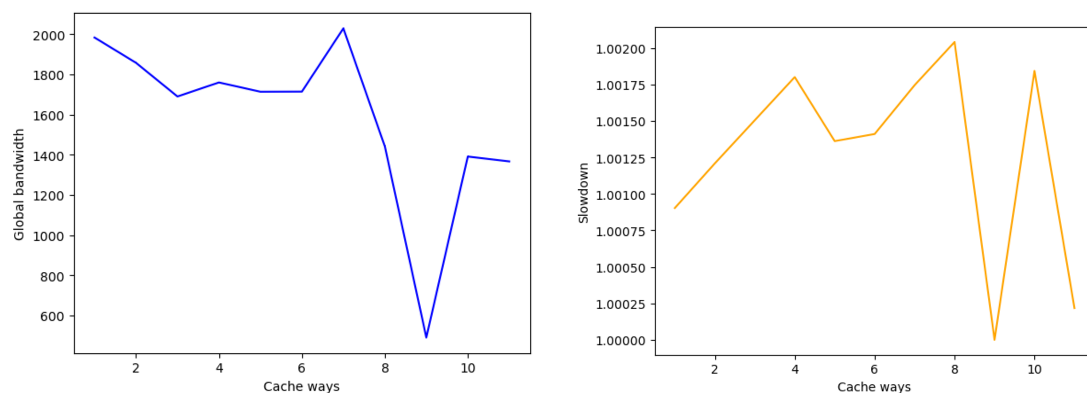


Figura 5.29: Gráfica de latencia de Data Serving Relational (modo OLTP-RW)

En la figura 5.29, se muestra la variación de la latencia en función de las vías de caché durante la ejecución de la aplicación Data Serving Relational en modo OLTP-RW. La gráfica revela que no hay incrementos significativos en la latencia; el pico más alto es de 20 ms cuando se utilizan 7 vías de caché, y en las demás configuraciones, la latencia se mantiene por debajo de 15 ms. Esto demuestra una latencia estable y baja, independientemente del número de vías de caché asignadas. Que indica que la aplicación no es particularmente sensible a la cantidad de caché disponible, reafirmando su clasificación como una aplicación de tipo Light Sharing.



(a) BW de Data Serving Relational (modo TPC-C)

(b) Slowdown de Data Serving Relational (modo TPC-C)

Figura 5.30: Gráficas de BW y Slowdown de Data Serving Relational (TPC-C)

En la figura 5.30, se presentan las gráficas de la media del ancho de banda y el tiempo de ejecución en función de las vías de caché disponibles para la aplicación Data Serving Relational en modo TPC-C. La aplicación muestra un bajo uso de

ancho de banda, y se observa que el aumento de las vías de caché no tiene un impacto significativo en su rendimiento. Aunque parece haber una ligera tendencia a la baja en el ancho de banda a medida que aumentan las vías de caché, esta tendencia es mínima. En cuanto al slowdown, es tan insignificante que puede ser despreciado. Basándonos en el comportamiento del ancho de banda y la falta de sensibilidad al cambio en las vías de caché, podemos clasificar la aplicación Data Serving Relational en modo TPC-C como un tipo de aplicación Light Sharing.

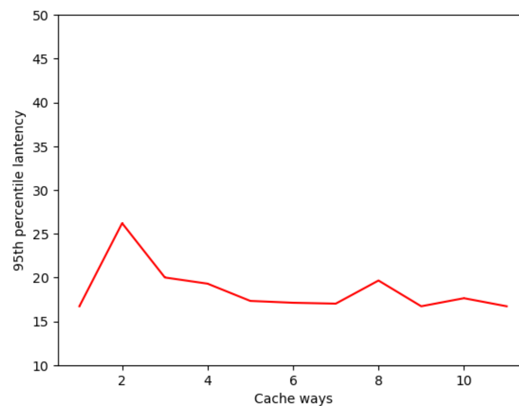


Figura 5.31: Gráfica de latencia de Data Serving Relational (TPC-C)

En la figura 5.31, se muestra la gráfica de latencia en función de las vías de caché disponibles para la aplicación Data Serving Relational en modo TPC-C. Observamos que, en general, no se observan incrementos significativos en la latencia. Aunque se identifica un pico de 25-30 ms con 2 vías de caché, esta latencia no es significativa en el contexto de esta aplicación. En las demás vías de caché, la latencia se mantiene consistentemente por debajo de los 25 ms.

Con la caracterización de aplicaciones realizada, hemos obtenido la clase de todos los *benchmarks* HPC y Cloudsuite seleccionados, siguiendo el contexto de la clasificación en tres grupos. Sin embargo, durante la clasificación de todas estas aplicaciones, hemos encontrado casos en los que estos tres grupos de clases se quedan cortos a la hora de caracterizar el comportamiento de estas. Esto se debe por ejemplo, a la existencia de aplicaciones con un gran consumo de ancho de banda cuyo rendimiento se ve bastante favorecido a más del tamaño de LLC que dispongan. Estas aplicaciones presentan rasgos de las clases cache-sensitive y streaming a la vez, lo que las hace difícil de catalogar. Esto mismo sucede con algunas aplicaciones las cuales requieren de un pequeño número de vías de caché para llegar a su funcionamiento aceptable, esto nos haría pensar en ellas como cache-sensitive, pero a pesar de esto una vez disponen de unas pocas vías su comportamiento pasa ser el que describiríamos en un programa light-sharing.

Es por todo esto, que además de obtener la caracterización de los distintos *benchmarks*, también surge de este análisis la idea de poder ser necesario definir nuevos grupos de clases para la correcta caracterización de los programas.

5.4.4. Experimentos con particionado de caché estático

Una vez clasificados los distintos *benchmarks* tanto cloud como HPC, hemos realizado una selección de un programa por clase tanto para HPC como cloud. Esto con el fin de lanzar mezclas de dos aplicaciones, compuestas por un programa HPC y otro Cloud, y así poder ver cómo afecta a estos el compartir la LLC y otros recursos compatibles en el mismo *socket* con programas de distintas clases. También se analiza el potencial del uso de particionado de cache para evitar la degradación de rendimiento entre programas. Además con estos experimentos, se pretende observar el impacto del particionado de caché a la contención de recursos.

5.4.4.1. Mezclas de aplicaciones

Las mezclas de aplicaciones que hemos escogido son:

- Aplicaciones Cloud:
  - **light-sharing**: data_serving_relational-oltp
  - **cache-sensitive**: graph_analitics-tc
  - **streaming**: graph_analitics-pr
- Aplicaciones HPC:
  - **light-sharing**: bodytrack
  - **cache-sensitive**: LU
  - **streaming**: BT

Experimentos	**Cloud**	**HPC**
exp1	postgresql-oltp	bodytrack
exp2	postgresql-oltp	LU
exp3	postgresql-oltp	BT
exp4	graph_analitics-tc	bodytrack
exp5	graph_analitics-tc	LU
exp6	graph_analitics-tc	BT
exp7	graph_analitics-pr	bodytrack
exp8	graph_analitics-pr	LU
exp9	graph_analitics-pr	BT

Tabla 5.2: Tabla de experimentos con mezcla de aplicaciones Cloud y HPC

Para los experimentos definidos en la tablas 5.2, se ha decidido las vías de cache que se asignaran los distintos programas siguiendo ciertos criterios. Los criterios que se han tenido en cuenta, han sido entre otros la clase de programa a la que

pertenecen, dando pocas vías a programas light-sharing ya que esto no les afecta en su rendimiento, así como dando pocas vías a streaming para que estos no contaminen la LLC al resto de aplicaciones. Además se ha tenido en cuenta el análisis del barrido de vías realizado en la sección 5.4.3.1. Esto con el fin de asignar el número de vías suficientes a los programas cache-sensitive, con el fin de evitar una degradación en su rendimiento por no contar con el número de vías suficientes. Por esto a la hora de mezclar una aplicación cache-sensitive con light-sharing o streaming, se ha decidido dar a estas últimas 2 vías de cache, ya que no es aconsejable dejar una sola vía para ningún programa como se menciona en[30], dejando 9 vías para las aplicaciones cache-sensitive.

Como último detalle antes de los análisis, cabe destacar que el tiempo de ejecución de los *benchmarks* de cloud del tipo cliente servidor, tienen un tiempo de ejecución fijo. Por tanto para poder medir como afecta a estos el particionado de cache, no se puede hacer uso del slowdown. Por esto para poder analizar el particionado de cache en estos se hace uso de la latencia. En nuestro caso esto afectaría a los experimentos exp1, exp2 y exp3 debido a que estas mezclas hacen uso de postgresql-oltp, para este programa se medira la latencia haciendo uso del percentil 95.

5.4.4.2. Comparación resultados con y sin particionado

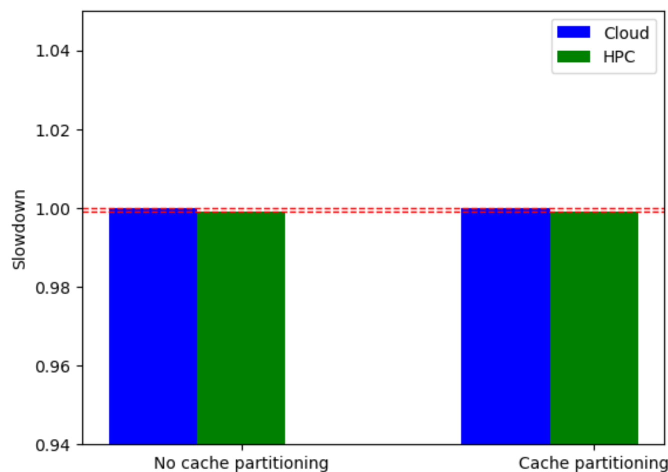


Figura 5.32: bodytrack vs postgresql-oltp

La figura 5.32 muestra la diferencia en slowdown para los programas bodytrack y postgresql-oltp. Para este experimento se le otorgó de 5 vías de cache a bodytrack y 6 vías para postgres-oltp. Estos apenas se ven afectado por el particionado de cache siendo imposible de percibir un cambio en el slowdown de estos tras el particionado. Esto se debe a que ambos son programas light-sharing, los cuales apenas consumen ancho de banda ni demandan uso de CPU, sumado a esto, este tipo de programas no se ve afectado por las vías de cache. Son todos estos motivos los que hacen que apenas haya cambios.

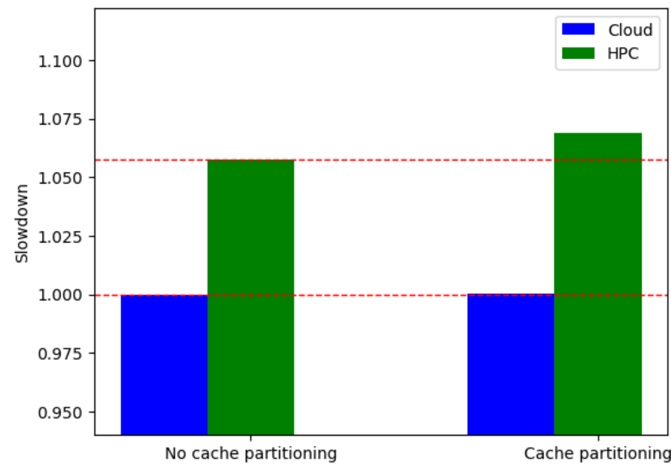


Figura 5.33: LU vs postgresql-oltp

En la figura 5.33, se muestra como afecta el particionado de cache para la mezcla de aplicaciones LU y postgresql-oltp mediante el slowdown. En estos resultados, podemos ver cómo apenas se producen cambios respecto al slowdown al hacer uso del particionado de cache. Al tratarse LU de un programa cache-sensitive, se le otorga a este 9 vías de cache, dejando a postgresql-oltp con 2 al ser light-sharing. Sin embargo, al ser nuestro *benchmark* cloud utilizado de tipo light-sharing, este apenas demanda uso de la cache lo que provoca que el particionado no afecte a el tiempo de ejecución de LU. Por último podemos ver que el particionado de cache no mejora el resultado porque hay un problema inherente de contención de recursos del bandwidth. Ambas apliaciones compiten por el ancho de banda, esto puede provocar la degradación que observamos.

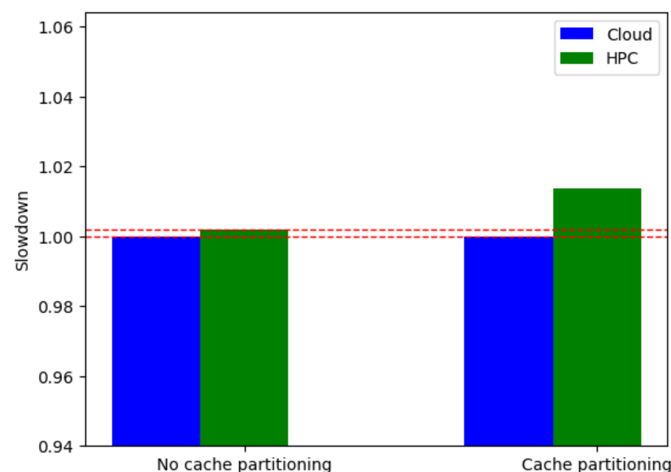


Figura 5.34: BT vs postgresql-oltp

Observamos la figura 5.34, la cual describe como afecta en el slowdown el uso de particionado para la mezcla de los programas BT con postgresql-oltp. En esta gráfica al igual que las otras podemos ver como para BT, el particionado de cache apenas causa cambios en su slowdown. Esto es debido a que se trata de un programa del

tipo streaming, el cual tiene un alto consumo de memoria cache compartida pero el cual su tiempo de ejecución apenas se ve afectado por el número de vías que dispone. Al tratarse de un programa streaming con alto consumo de ancho de banda, se le otorgó a este de 7 vías de cache, dejando las 4 restantes para postgresql-oltp.

Para este primer conjunto de experimentos, al haberse hecho uso de un *benchmark* de cloud del tipo cliente servidor, para este no se puede medir como afecta el particonado de cache mediante el slowdown. Por esto, también se realizó un análisis de la latencia en percentil 95 de postgresql-oltp para los tres experimentos.

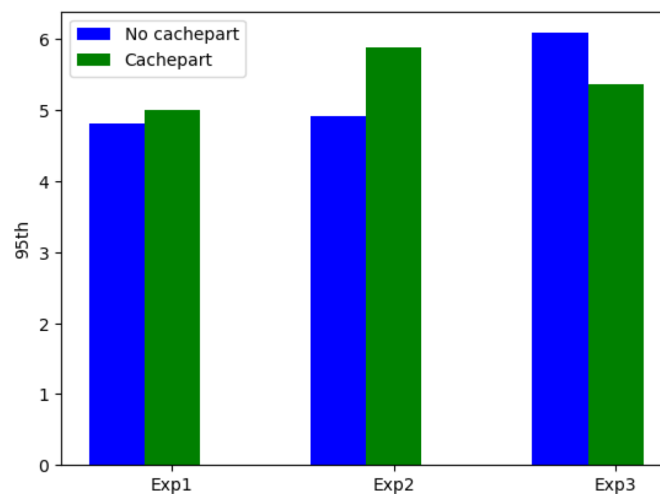


Figura 5.35: Comparación de latencia del *benchmark* postgresql-oltp en los exp 1, 2 y 3

Como se puede observar en la figura 5.35, la cual muestra la variación de la latencia frente al uso de particionado de cache para los tres experimentos de postgresql. La latencia apenas presenta cambios, como cabe de esperar en un programa del tipo light-sharing. Esta latencia presenta su mayor subida en el experimento 2 ya que para este se limitan sus vías de cache a 2 vías, esto nos muestra como la contención de recursos puede llegar a afectar de manera mínima la latencia de este. En el experimento 3, se presenta el caso contrario, al encontrarse compartiendo CPU junto con un programa de tipo streaming, la latencia presenta cierta mejora al usar particionado de cache, ya que este particionado limita el uso de memoria cache de tercer nivel que puede hacer uso BT, evitando que este afecte a la latencia de postgresql. En cualquier caso, no se viola en ningún momento los requisitos de latencia de este programa cloud.

Por lo general, vemos en estos tres experimentos, como los programas light-sharing, apenas se ven afectados por el resto de programas, ya sea con o sin particionado de cache, llegando a apenas verse afectados por programas del tipo streaming.

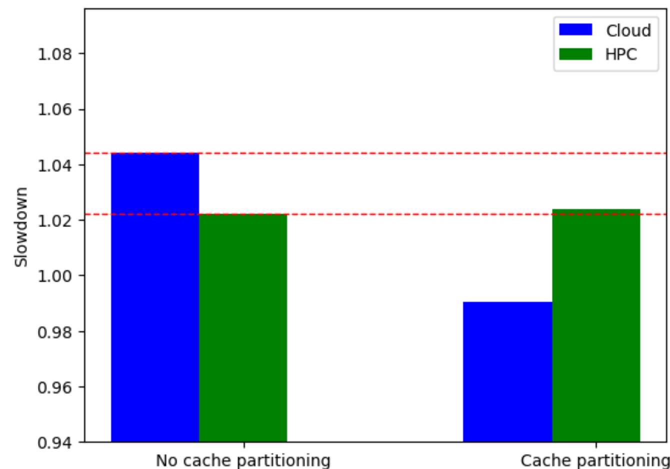


Figura 5.36: bodytrack vs graph_analytics-tc

En la figura 5.36, podemos observar cómo afecta el partionado de cache a la mezcla de programas bodytrack con graph_analytics tc. Para este experimento, graph_analytics tc dispone de 9 vías de cache al tratarse de un programa cache-sensitive, las 2 vías restantes son usadas por el programa HPC light-sharing bodytrack. Como se puede observar en los resultados, se percibe una leve mejora en el slowdown del *benchmark* cloud, esto es debido a que se trata de un programa cache-sensitive. Sin embargo, esta se presenta de forma tan leve debido a encontrarse junto a un programa light-sharing, el cual apenas demanda uso de la LLC.

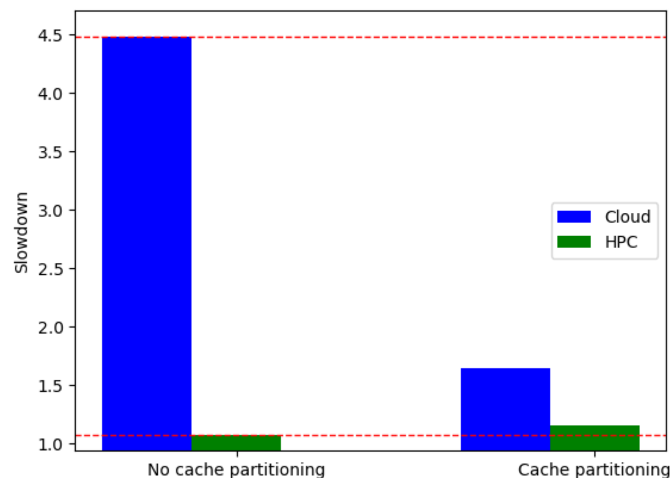


Figura 5.37: LU vs graph_analytics-tc

La figura 5.37, describe los resultados del particionado de cache para la mezcla de LU con graph_analytics-tc. Para esta mezcla, se particionó la cache dando 5 vías a LU y 6 a graph_analytics-tc. Se puede observar como el particionado de cache ha mejorado significativamente el rendimiento de nuestro *benchmark* cloud, y apenas degradando el rendimiento del programa HPC. Estos resultados, se producen debido a que este experimento combina dos programas cache-sensitive. Al no realizar un particionado de la cache, ambos se encuentran demandando gran cantidad de

espacio en la LLc, lo cual afecta al rendimiento de ambos. Al aplicar particionado de cache, nos aseguramos que estos sólo hagan uso de la porción de cache asignada sin interferir en el otro, lo que se traduce en la gran mejora de rendimiento reflejada en el slowdown de graph_analytics-tc. También podemos ver como enpeora levemente el rendimiento de LU, esto se debe a problemas de contención en el ancho de banda, los cuales no se pueden solucionar aplicando solo particionado de cache.

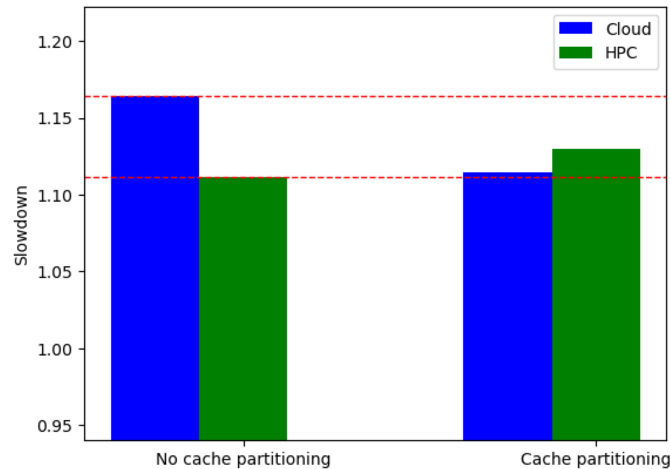


Figura 5.38: BT vs graph_analytics-tc

En la figura 5.38, se muestra como afecta al slowdown el particionado de cache de la mezcla de aplicaciones de BT y graph_analytics-tc. En este experimento, se le ha otorgado 8 vías de cache a graph_analytics-tc y las 3 restantes a BT. En los resultados obtenidos, se puede ver una pequeña mejora con respecto al slowdown en nuestro programa cloud al aplicar el particionado. Sin embargo, esta no es una mejora muy grande para tratarde de un particionado de una aplicación cache-sensitive frente a una streaming y cabría esperar una mejora significativa. Este problema puede deberse a la gran cantidad de ancho de banda que consume BT, el cual puede acabar en la pérdida de rendimiento de graph_analytics-tc a pesar de hacer uso del particionado, lo que explica la baja mejora producida.

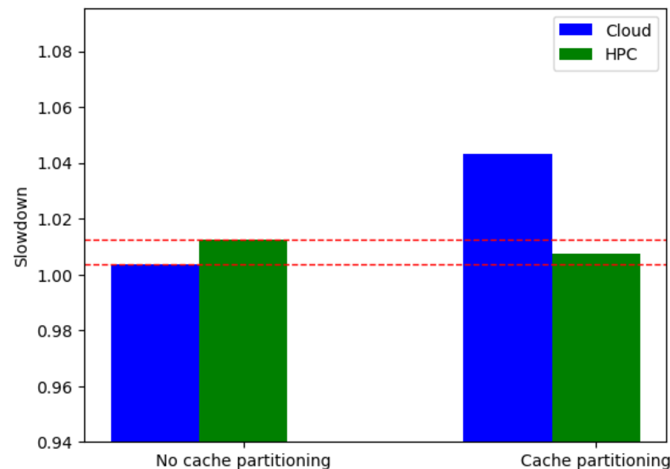


Figura 5.39: bodytrack vs graph_analytics-pr

La figura 5.39, nos muestra como afecta el particionado de cache a la mezcla de bodytrack con graph_analytics-pr. En este experimento, graph_analytics-pr dispone de 7 vías de cache y bodytrack hace uso de las 4 restantes. En estos resultados podemos ver como bodytrack no se ve afectado por el particionado de cache, como cabe esperar de un programa light-sharing. Por otro lado, graph_analytics-pr presenta una pequeña subida del slowdown al hacer uso del particionado, esto puede deberse debido a que para este programa la contención de recursos llega a afectar en su rendimiento. A pesar de esto, al tratarse de un programa streaming, esta subida del slowdown es ínfima y apenas remarcable.

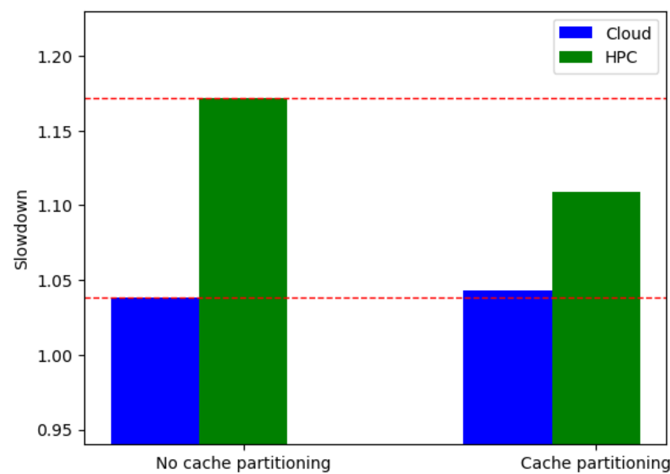


Figura 5.40: LU vs graph_analytics-pr

Como podemos observar la figura 5.40 muestra los resultados del particionado de la cache para el conjunto de aplicaciones LUy graph_analytics-pr. En esta carga, el particionado de cache consigue una ligera mejora en el rendimiento de LU, al cual se le otorgan 9 vías de cache; las 2 vías restantes son asignadas a graph_analytics pr. Esta ligera mejora cabría esperar que fuera mayor al tratarse de un programa HPC cache-sensitve junto con un programa cloud streaming, al limitar el uso de cache de

tercer nivel que hace el programa streaming. La explicación de la pequeña mejora, la podemos encontrar en el gran uso de ancho de banda que hace graph_analytics-pr el cual termina por afectar al rendimiento de nuestro *benchmark* HPC.

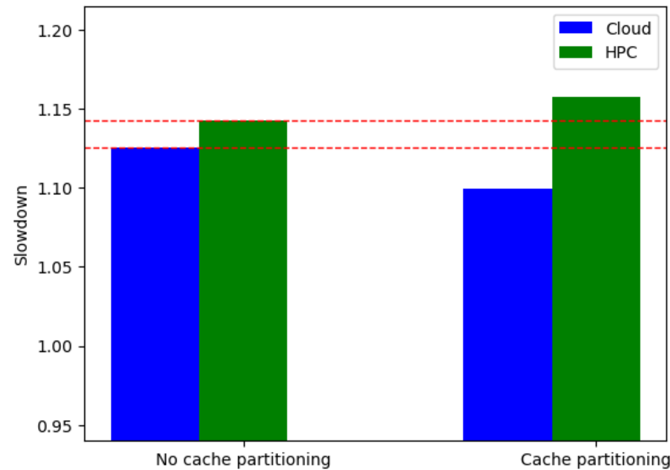


Figura 5.41: BT vs graph_analytics-pr

La figura 5.41, muestra como afecta el particionado de cache a la mezcla de los *benchmarks* BT y graph_analytics-pr. Para este experimento, graph_analytics-pr dispone de 7 vías de cache y BT de 4 vías. Como observamos en los resultados, el particionado produce una leve mejora en graph_analytics-pr y una ínfima subida del slowdown en BT. Esto ocurre al tratarse de dos programas del tipo streaming, los cuales apenas se ven afectados por el tipo de vías, a todo esto se le suma el gran consumo de ancho de banda que poseen ambas aplicaciones. Todo esto hace difícil que el particionado de cache produzca una mejora.

Como conclusión final, a través de estos experimentos de llega a demostrar como el particionado de cache de forma estática puede no incrementar la degradación del rendimiento en los programas. También se observa cómo las distintas mezclas de clases de programas se ven afectadas de forma distinta y cómo dependiendo de la mezcla y el tipo de clase de la aplicación estas se benefician más o menos del particionado, llegando a ser los programas cache-sensitive los que mayor beneficio reciben al aplicarse el particionado de cache correcto sobre estos. Por último estos experimentos también arrojan la idea de que solo aplicando particionado de cache, no puede garantizarse un correcto aislamiento entre aplicaciones, especialmente cuando estas son intensivas en ancho de banda.

Capítulo 6

Extensión del sistema operativo para clasificación de aplicaciones

En este capítulo se describen los distintos pasos que se han seguido para la implementación de un *plugin* de clasificación de aplicaciones de PMCSChed.

6.1. Análisis experimental y heurística

Para poder conocer la clase de un programa, uno de los métodos a seguir es ejecutar de este con distintos número de vías de la LLC y ver los cambios que se producen en el tiempo de ejecución, el consumo de ancho de banda y/o la latencia. Este método presenta un inconveniente: esto solo se puede realizar de forma *offline*. Sin embargo, existen otros métodos de clasificación de programas los cuales se realizan de forma dinámica, es decir, en tiempo de ejecución del programa. Un ejemplo de estos métodos de caracterización, es el que implementa la estrategia de particionado de cache LFOC+ [30]. Esta de forma dinámica, clasifica los distintos programas haciendo uso del particionado de cache. Para ello hace uso de una partición de cache, la cual reserva llamada *sampling partition*. Cuando no se tiene registro previo de su clase, este es asignadi a la *sampling partition*. En ella se le va asignando un mayor número de vías al programa de forma periódica y se observa como esta afecta al programa hasta que se conoce su clase. En ese momento la caracterización finaliza y este programa es recolocado siguiendo la política de particionado impuesta por el algoritmo subyacente.

Sin embargo, este método de caracterización es algo costoso debido a que requiere de decenas de milisegundos para caracterizar una aplicación, y esto depende del número de vías que otorgar a la aplicación hasta saber su clase. Otro problema que presenta es la reserva de parte de la cache de tercer nivel para crear la *sampling partition*. Debido a esto, algunas investigaciones plantean la idea de medir el tráfico entre los distintos niveles de cache para así determinar la clase de los programas

[10]. Esto plantearía una solución al coste del uso de una partición de cache para la caracterización. El problema con este método de caracterización, se encuentra en que no tenemos con los contadores hardware (PMCs), una forma de medir el tráfico entre los distintos niveles de cache en sí mismos. Además en [10] no se proponen valores umbrales, mediante los cuales poder distinguir entre una clase u otra.

Por todo esto, para poder hacer posible una clasificación en nuestras plataformas se hizo una selección de métricas candidatas, las cuales pueden ser calculadas a través de los PMCs de la CPU. Para poder obtener el valor de las métricas de distintos programas, se realizó un *plugin* de PMCSched 4.

Nombre	**Evento que definen**
instr	Número de instrucciones
cycles	Número de ciclos
llc_references	Número de referencias a la LLC
llc_misses	Número de fallos en la LLC
l1d.replacement	Número de remplazos en la cache L1
l2_lines_in.all	Número de líneas de L2 ocupadas

Tabla 6.1: PMCs usados para el calculo de métricas

Con eventos que se observan en la tabla 6.1 configurados en nuestro *plugin* de PMCSched, podemos sacar los valores de las siguiente métricas:

Métrica	**Fórmula**	**Defición métricas**
ipc	$instr/cycles$	Instrucciones por ciclo
llcrpki	$(llc_references * 1000)/instr$	Referencias a la LLC cada 1000 instrucciones
llcmpki	$(llc_misses * 1000)/instr$	Fallos a la LLC cada 1000 instrucciones
llcrpkc	$(llc_references * 1000)/cycles$	Referencias a la LLC cada 1000 ciclos
llcmpkc	$(llc_misses * 1000)/cycles$	Fallos a la LLC cada 1000 ciclos
stalls_l2_miss	$l1d.replacement/cycles$	Ciclos de parada causados por fallos en la L2
llcm_bandwidth	$(llc_misses * 64)/etime_us$	Consumo ancho de banda de la LLC
l1bw	$(l1d.replacement * 1000)/instr$	Consumo ancho de banda de la L1
l2bw	$(l2_lines_in.all * 1000)/instr$	consumo ancho de banda de la L2
l1reuse	$l1d.replacement/l2_lines_in.all$	Reuso de la cache de primer nivel
l2reuse	$l2_lines_in.all/llc_misses$	Reuso de la cache de segundo nivel

Tabla 6.2: Métricas usadas

Algunas de las métricas que encontramos en la tabla 6.2 se usan mucho en investigación sobre arquitectura de computadores, ya que se tratan de gráficas muy estables. Estas métricas fueron escogidas debido a que sin medir puramente el tráfico entre los distintos niveles de cache, el valor de estas está relacionado con el uso que hace los distintos programas de la jerarquía de cache[1].

Una vez definidas las métricas, las cuales vamos a explorar su uso para la clasificación de aplicaciones, se realiza un barrido de vías para un conjunto de *benchmarks* HPC los cuales ya habían sido caracterizados [21]. En nuestro caso este experimento es realizado en nuestra plataforma Broadwell la cual se describe en la sección 5.1.1, y este barrido se hace desde 1 vía de cache hasta 20.

Con este experimento se pretende ver el valor de estas métricas al variar gradualmente la ocupación de cache para las distintas clases de aplicaciones. Esto con el fin de encontrar unos valores umbrales con los cuales poder formar una heurística que pueda ser usada para la caracterización de programas.

Una vez realizado el experimento, se realiza un análisis de los resultados obtenidos. Primero, se construye una matriz de correlación entre las distintas métricas, esto con el fin de observar si algunas de estas métricas se encuentran fuertemente correlacionadas y por tanto, nos podemos quedar con una sola de ellas, reduciendo así el número de métricas del análisis.

[1] etime_us es una métrica que nos proporciona la propia herramienta PMCTrack, y que mide el tiempo transcurrido entre dos muestras.

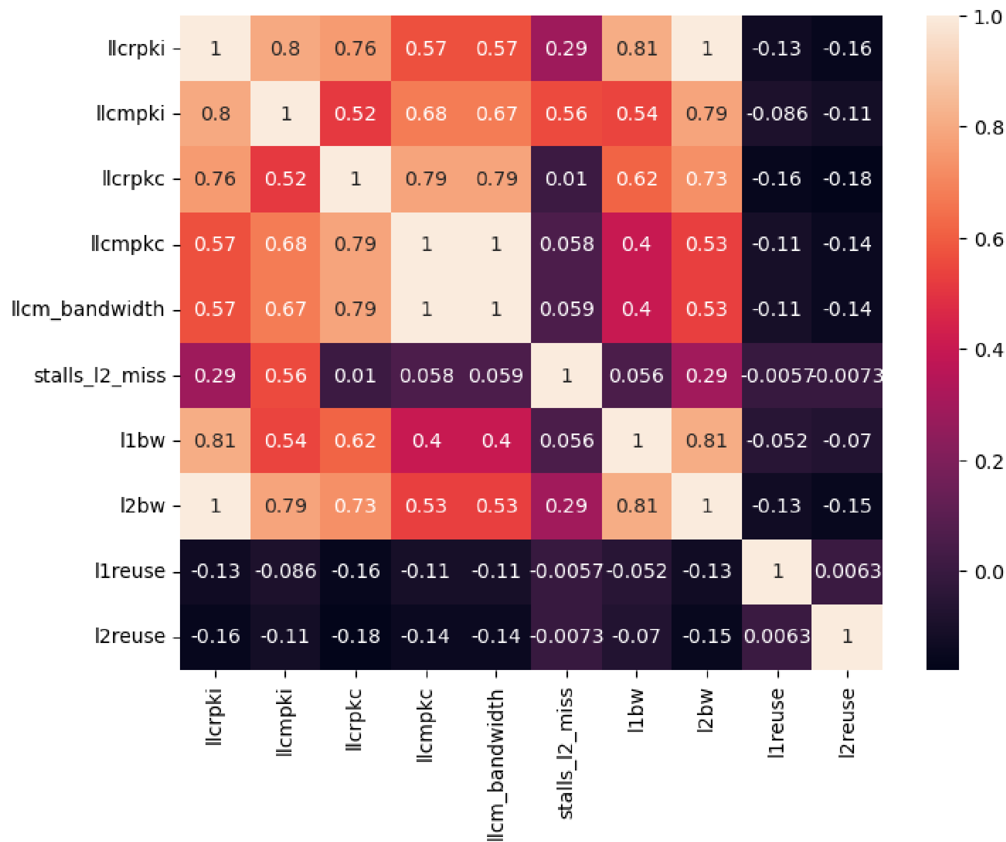


Figura 6.1: Matriz de correlación para las métricas

La figura 6.1 nos muestra la matriz de correlación obtenida para las métricas. En los resultados de esta, observamos que hay varias métricas que están fuertemente relacionadas entre sí, y que por tanto podemos quedarnos solo con una de ellas. Por esto, se elimina de nuestro análisis las métricas de l2bw y llcm_bandwidth. Además se elimina también IPC (*instructions per cycle*) considerando que esta no juega un papel fundamental en el tipo de comportamiento de los programas.

Una vez disponemos de todas las métricas que finalmente serán usadas y para tener un acercamiento de cuáles de estas métricas son posibles candidatas en lo que respecta a la clasificación de programas. Se hace uso del método de machine learning del árbol de decisión. Este mecanismo de machine learning es interesante debido a su uso en problemas de clasificación y porque este nos proporciona como resultado un árbol binario, el cual sirve para clasificar mediante las métricas. Lo más interesante y el principal motivo de hacer uso de este método de machine learning, se debe al uso de la entropía para construir los distintos nodos del árbol, siendo el nodo de más arriba el que mayor ganancia de información nos aporta y por tanto un posible candidato a tener en cuenta para la clasificación de programas.

La figura 6.2 muestra la matriz de confusión obtenida tras probar la clasificación obtenida mediante el árbol de decisión. Los datos revelan que el algoritmo de machine learning del árbol de decisión ha obtenido muy buenos datos de predicción. Esto nos

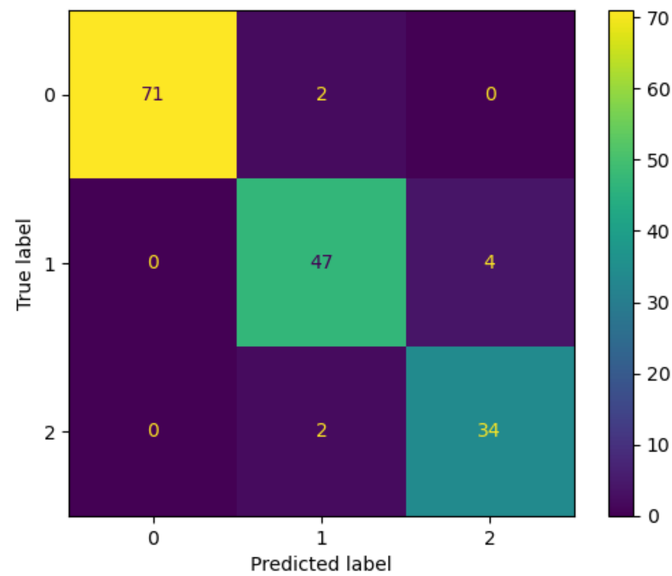


Figura 6.2: Matriz de confusión de la predicción del árbol de decisión

indica que podemos tomar bastante en consideración los datos arrojados por el propio árbol, además de indicarnos que el uso de estas métricas para la clasificación de programas parece no ser una mala idea.

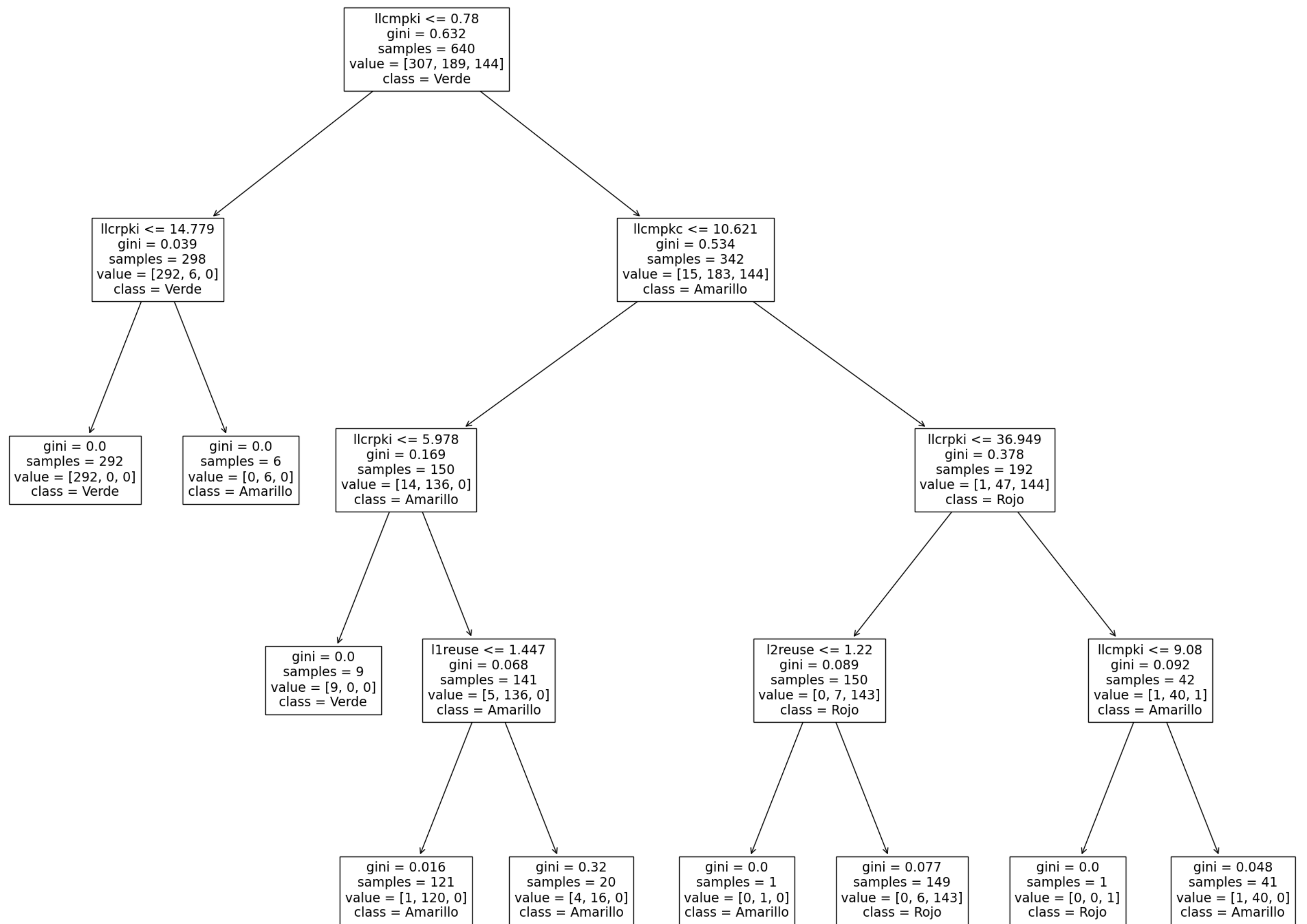


Figura 6.3: Árbol de decisión de altura 4

En la figura 6.3, que muestra el árbol de decisión obtenido, podemos ver que el nodo raíz lo ocupa la métrica llcmpki. Esto nos indica que esta es la métrica que más agrupa los datos para la clasificación. Al haber limitado la altura del árbol a un máximo de 4, podemos tomar todas las métricas que aparecen en este como posibles candidatas a la hora del estudio de posibles umbrales de métricas para la clasificación de programas.

Con los buenos resultados obtenidos a través del árbol de decisión y para seguir comprobando la posibilidad de clasificar los programas a partir de las métricas, se realiza una gráfica de la distribución espacial de los distintos programas. Esto nos dará una mejor perspectiva para ver si las distintas clases se encuentran agrupadas entre si o sin embargo no se distinguen grupos.

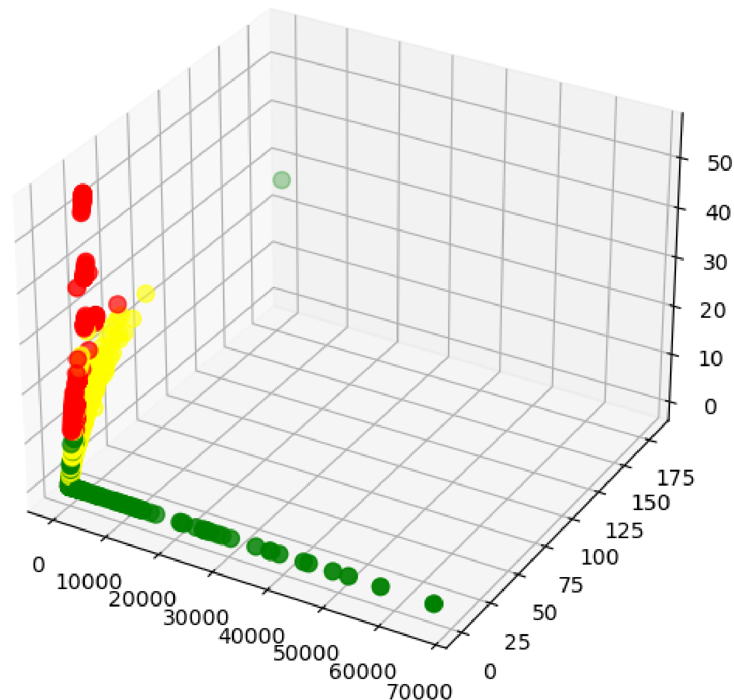
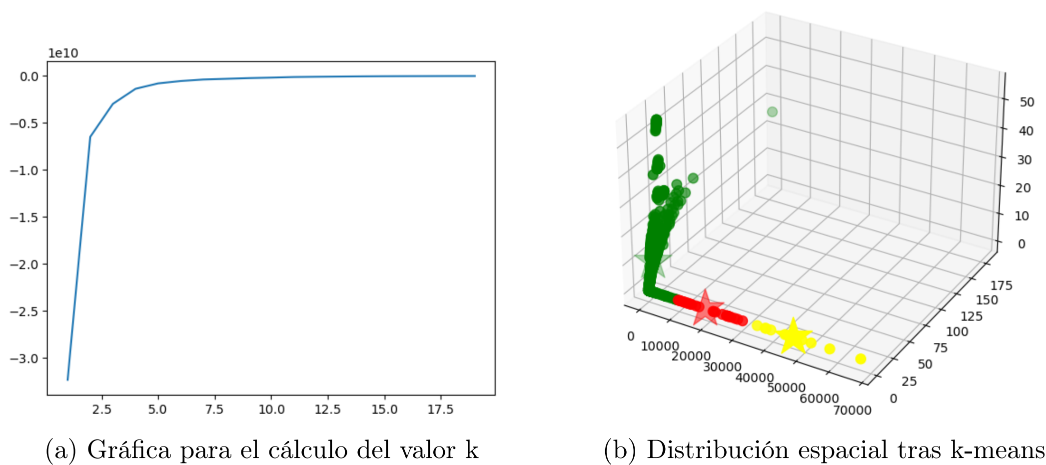


Figura 6.4: Distribución espacial de las distintas clases

La figura 6.4 presenta la distribución espacial de las distintas clases según sus métricas, se puede observar como las clases se encuentran bastante agrupadas. Sin embargo entre las clases streaming (rojo) y cache-sensitive (amarillo) los grupos se encuentran muy cercanos entre sí, lo cual nos indica que puede haber problemas a la hora de intentar distinguir aplicaciones de estas dos clases. También encontramos unos pocos casos en los que programas light-sharing (verdes) se agrupan junto con algunos cache-sensitive.

Continuando con el esfuerzo por conocer si con las métricas y las clases que tenemos podemos agrupar correctamente entre estas, se procede a aplicar otro algoritmo de machine learning, el cual es especialmente usado para casos como este en los que queremos agrupar nuestras distintas clases. Se trata de k-means, que a través de un valor k, fijado antes de ejecutar el algoritmo, agrupa en k grupos los distintos objetos que se le pasan, en nuestro caso los programas. Esto lo hace basándose en las características de cada uno.



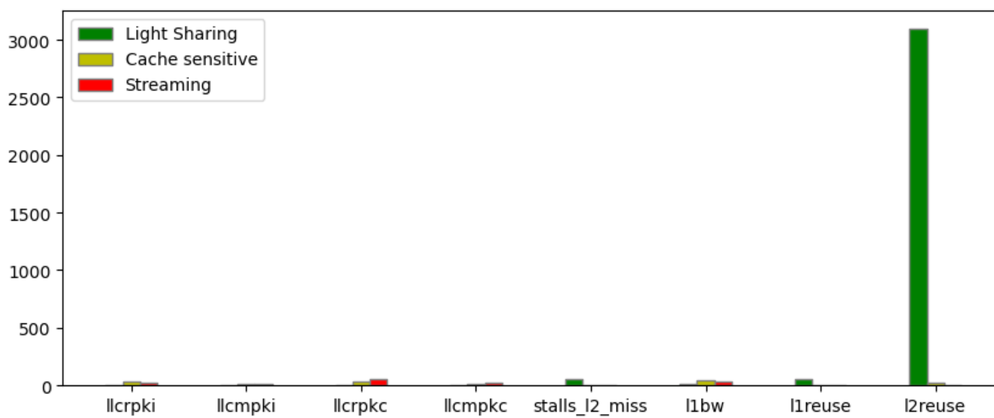
(a) Gráfica para el cálculo del valor k

(b) Distribución espacial tras k-means

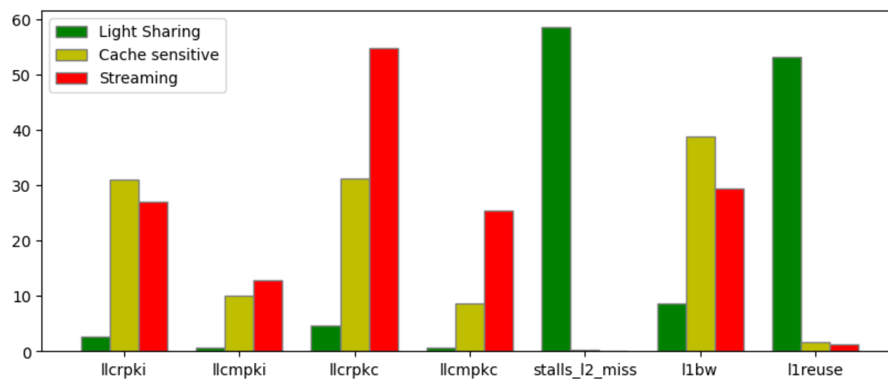
Figura 6.5: Resultados k-means

La figura 6.5a describe la gráfica del método del codo para el cálculo de la k. Este consiste en trazar la suma de las distancias al cuadrado entre cada punto de datos y su *centroide* asignado para diferentes valores de k. En este método, buscamos el valor de k donde la disminución en la suma de las distancias al cuadrado se ralentiza y forma una curva similar a un codo. Los resultados revelan que puede hacer uso de una k entre 3 y 5, lo cual nos dice que tres clases parece un buen acercamiento pero que quizá se pueden agrupar en alguna más. Finalmente, se hace uso de una k de valor 3 debido a que originalmente disponíamos de tres tipos de clases entre las que queremos clasificar: light-sharing, cahce . Habiendo hecho uso de tres agrupaciones para nuestro algoritmo, obtenemos la figura 6.5b la cual muestra la distribución espacial de los datos tras ejecutar K-means para k=3. Se obtienen muy buenos resultados teniendo ahora agrupaciones mucho mejor definidas y sin mezclas entre clases.

Con estos resultados podemos seguir con el análisis de los valores de la métricas de las distintas clases sabiendo que este es un buen acercamiento para la clasificación. Para esto, vamos a analizar los valores de las distintas métricas para cada clase empezando por la media de las métricas.



(a) Medias de todas las métricas

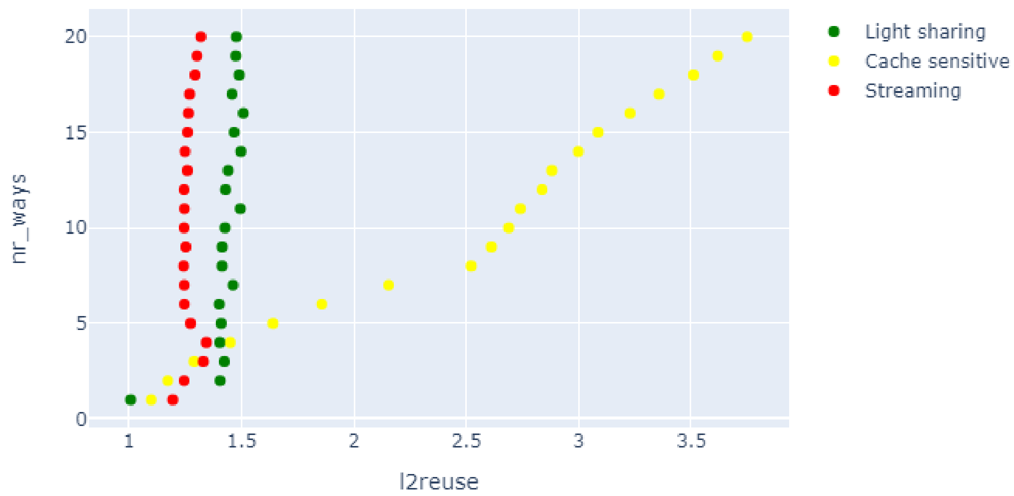


(b) Medias sin l2reuse

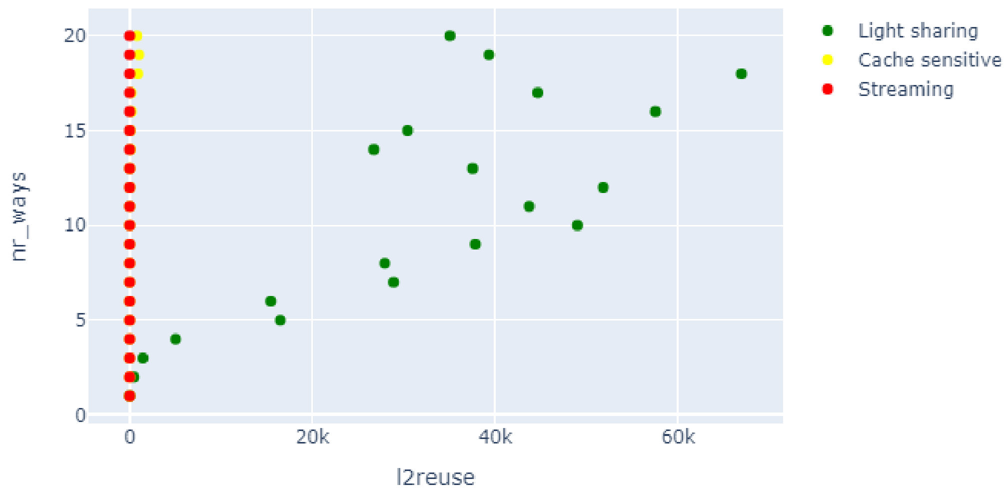
Figura 6.6: Medias de las métricas según su clase

La figura 6.6, muestra la media de las tres clases para todas las mértricas. Podemos ver como la media para los programas light-sharing de l2reuse es mucho mayor a la de las otras dos clases. Esto nos muestra un claro candidato para la clasificación de programas light-sharing. Además también podemos ver que con respecto a las medias parece ser que cualquier media parece buena para clasificar programas light-sharing. Las métricas que aparentemente pueden ser buenas candidatas para clasificar los prgramas cache-sensitive y streaming parecen poder ser llcrpkc y llcmpkc, ya que estas métricas son las que muestran métricas más dispares entre todas las clases.

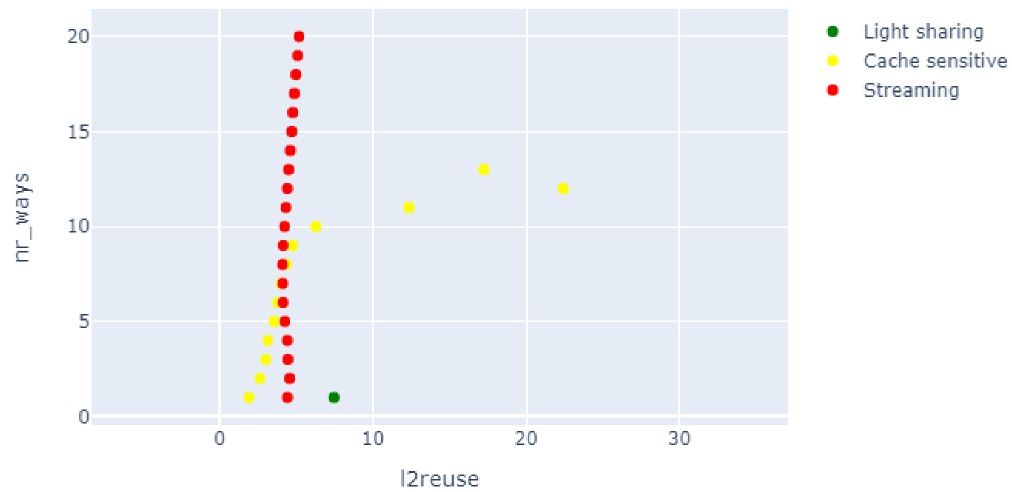
Tras tener claro los posibles candidatos y ver que claramente las métricas muestran diferencias entre las distintas clases, el siguiente paso es sacar valores umbrales para algunas métricas que nos permitan clasificar las distintas clases. Para esto se ha de evaluar a través de los distintos números de vías y ver los valores umbral de las distintas métricas (máximos y mínimos) para cada clase.



(a) Mínimos l2reuse según vías de cache



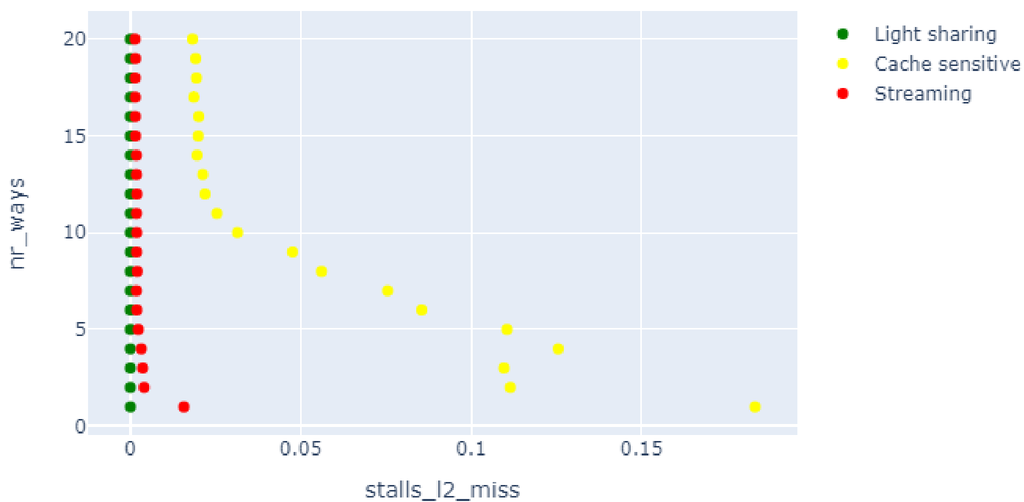
(b) Máximos l2reuse según vías de cache



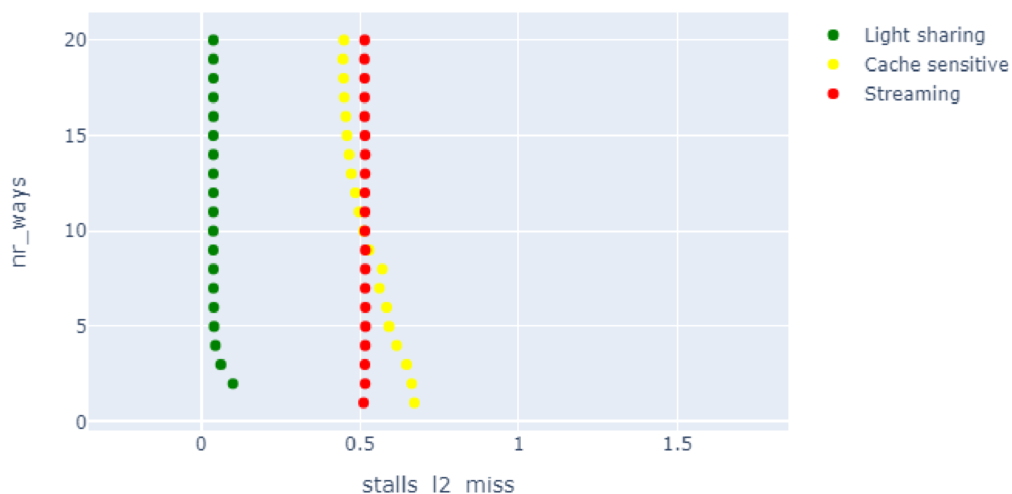
(c) Máximos l2reuse según vías de cache (cache-sensitive y streaming)

Figura 6.7: Máximos y mínimos l2reuse según vías de cache

Tras analizar los valores máximos y mínimos de l2reuse según el número de vías como se muestra en la figura 6.7, llegamos a la conclusión de que a pesar de su alta media en programas light-sharing, no se puede poner un umbral mínimo de esta para caracterizar las aplicaciones de esta clase debido a que estas pueden llegar a tener valores mínimos menores a los de aplicaciones cache-sensitive. Sin embargo, los valores máximos para light-sharing pueden ser una buena medida para clasificar programas light-sharing ya que esta clase alcanza valores muchísimo más grandes para esta métrica que el resto de clases. Por último, los resultados sugieren que esta métrica puede ser interesante a la hora de poder clasificar programas streaming, ya que estos muestran unos máximos para esta métrica bastante menores a los del resto clases a partir de 9 o 10 vías.



(a) Mínimos stalls_l2_miss según vías de cache



(b) Máximos stalls_l2_miss según vías de cache

Figura 6.8: Máximos y mínimos stalls_l2_miss según número de vías

Observando los resultados para los valores máximos y mínimos de la métrica stalls_l2_miss según el número de vías que se describen en la figura 6.8. Los programas light-sharing son aquellos que en sus máximo y mínimo tienen los valores menores con respecto a las otras clases. Sin embargo, los programas streaming en su mínimo su valor para esta métrica es menor que el máximo de la clase light-sharing. También se puede observar como efectivamente los programas cache-sensitive son mucho más irregulares sus valores en las distintas métricas a medida que estos aumentan su número de vías.

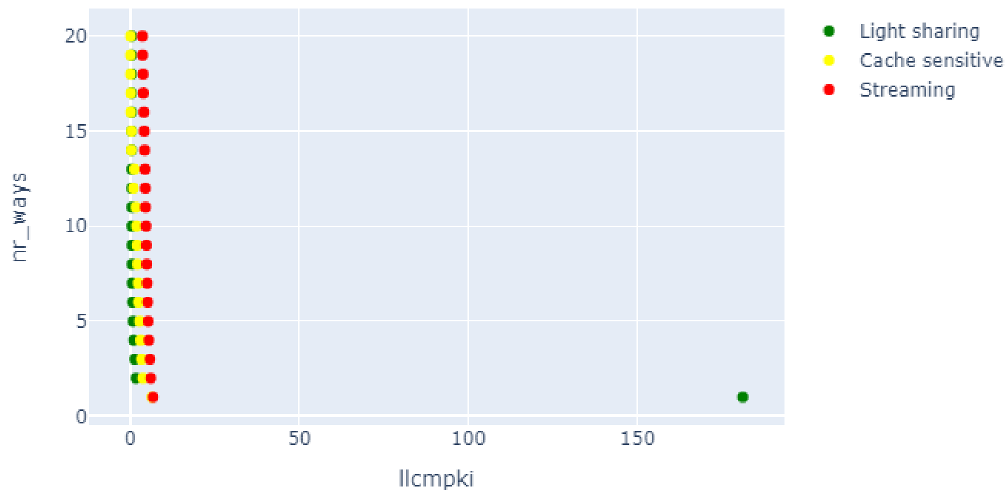


Figura 6.9: Máximos light-sharing frente a mínimos aplicaciones cache-sensitves y streamings llcmpki

Mediante llcmpki nos encontramos la primera métrica que nos da un umbral bastante claro, como nos muestra la figura 6.9 en los máximos para light-sharing y mínimos del resto de clases para llcmpki. En este caso, para la clasificación de aplicaciones light-sharing. Esto se debe a que esta métrica para la mayoría de número de vías, se encuentra en su valor máximo, siendo menor al mínimo del resto de clases. Gracias a esto podemos escoger un valor umbral a partir del cual si este es menor podemos saber con certeza que estamos ante un programa light-sharing. Además en este caso observamos un detalle importante, que ya se ha podido observar en otras métricas, como se observa en la figura 6.7c. Las distintas clases presentan un comportamiento extraño cuando estas poseen una vía de cache.

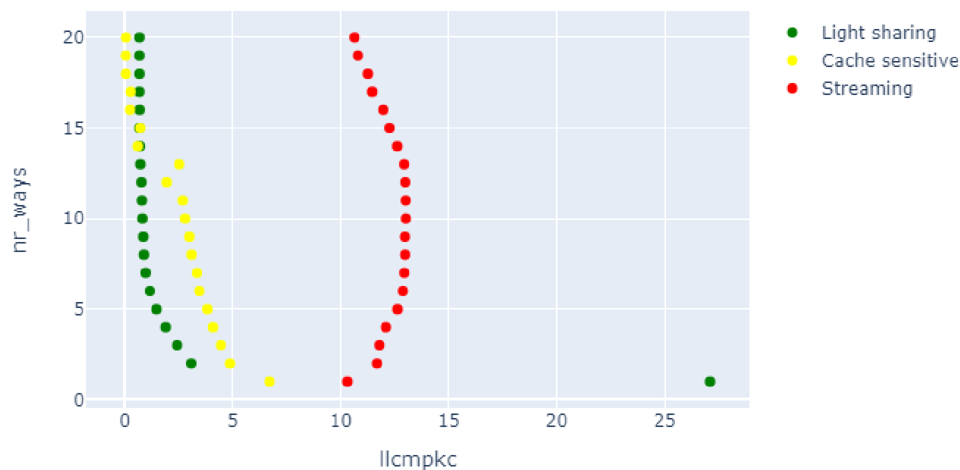


Figura 6.10: Máximos light-sharing frente a mínimos cache-sensitves y streamings llcmpkc

Al igual que pasaba con la métrica llcmpki, llcmpkc nos da un umbral claro que aplicar para clasificar las aplicaciones light-sharing como se observa en la figura 6.10. El máximo de esta métrica para los programas light-sharing, hasta un máximo de unas 14 vías, es menor a los mínimos de las dos clases restantes. También esta gráfica nos vuelve a mostrar el problema de intentar caracterizar programas con una sola vías como se observa con el valor para una vía del mínimo de llcmpkc para los programas light-sharing.

Al finalizar el análisis de todas la métricas, nos encontramos con buenos resultados en lo que respecta a la caracterización de programas light-sharing pero sin embargo, no contamos con una forma clara haciendo uso de las métricas, de caracterizar entre programas streaming y cache-sensitive. Es por eso que se realizó un último análisis. Para ello se crea una nueva métrica la cual es la diferencia entre llcrpkc y llcmpkc, esto debido a que ambas clases tienen un ratio muy alto de accesos a las LLC pero sin embargo son los programas streaming los que más fallos acumulan en la LLC sin importar el número de vías. Es por todo esto que la creación de esta nueva métrica nos puede dar una buena referencia para diferenciar entre estas dos clases.

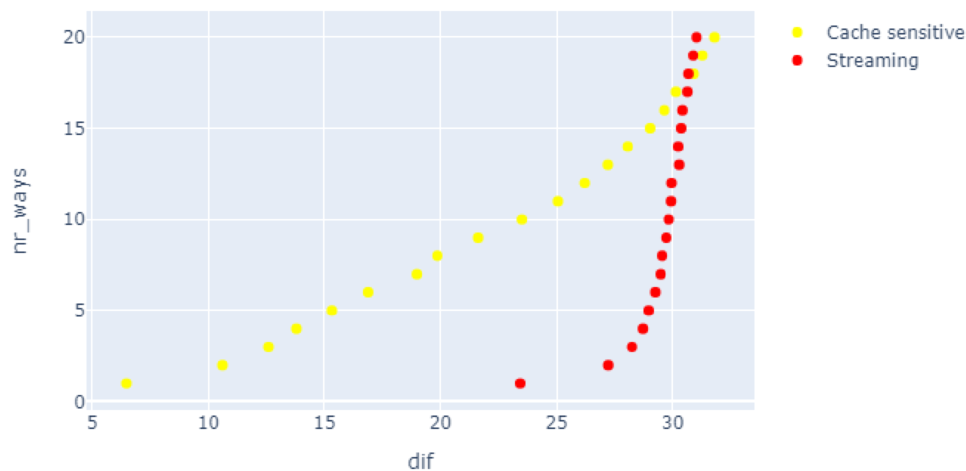


Figura 6.11: Media de la diferencia entre llcrpkc y llcmpkc

Como podemos observar en la figura 6.11, la cual muestra la media según el número de vías de la diferencia entre entre llcrpkc y llcmpkc para clases streaming y cache-sensitive. Esta nueva métrica no se comporta nada mal a la hora de distinguir entre programas cache-sensitive y streaming. Sin embargo, esta sigue teniendo problemas a la hora de mirar casos extremos como son los máximos y los mínimos, donde nos podemos seguir encontrando casos que no cumplen que esta métrica sea siempre menor en los cache-sensitive, sin importar el número de vías. Aún así, de las métricas analizadas en lo que respecta a sus medias, esta es la que en términos de media mejor se distingue entre estas dos clases y por tanto nuestro mejor candidato para su uso en la heurística de clasificación.

Finalmente y tras haber realizado un análisis exhaustivo de los resultados obtenidos, se desarrolló una heurística para clasificación de aplicaciones. Para esta hemos tenido en cuenta métricas como pueden ser llcmpkc y llcmpki debido a los buenos datos arrojados con respecto a su uso para la clasificación de programas light-sharing. Para la caracterización de aplicaciones cache-sensitive y streaming, debido a la dificultad que esto plantea con las métricas escogidas, se ha hecho una selección de varias de estas métricas, siendo estas: l2_reuse, stalls_l2_miss y la métrica creada a partir de la diferencia de llcrpkc y llcmpkc; quedando heurística de la siguiente forma:

```
Si(llcmpkc <= 1,8 o llcmpki <= 3,25)
    clase = light-sharing
Si no{
    si (stalls_l2_miss <= 6,5){
        clase = streaming
        }
    sino si (stalls_l2_miss > 0,005){
        clase = cache-sensitive
        }
    sino si (l2_reuse > 5,00){
        clase = cache_sensitive
    }
    sino si ( diff <= 17,00){
        clase = cache-sensitive
    }
    sino{
        clase = streaming
        }
    }
```

[2]

Cabe destacar que durante el análisis de todas estas métricas obtenidas, no se ha podido obtener un valor umbral para la clasificación de cada clase que clasifi-case correctamente en todos los casos. Se ha tratado de implementar la heurística más acertada con los resultados obtenidos, sin embargo, esta está sujeta a fallos de predicción sobre todo entre programas cache-sensitive y streaming.

Más tarde se realiza una evaluación para esta eurística sobre un conjunto de *benchmarks*.

6.2. Plugin de caracterización de aplicaciones

En conjunto con la clasificación de programas cloud y HPC mediante barrido de vías, se realizó otra clasificación mediante el uso de la heurística desarrollada. Esto con el fin de primero, comprobar el funcionamiento del método de clasificación desarrollado, y segundo para usar esta como ayuda a la hora de definir las clases de los programas de los cuales no teníamos registro de su clase.

Para lograr esto, lo primero que se necesita es un *plugin* de PMCSched el cual implementa nuestra heurística y guarda registro de la clase de las distintas aplicaciones que se lancen. Para ello se crea un nuevo *plugin* de PMCSched.

[2] La métrica diff del pseudo código se refiere a la diferencia entre llcrpkc y llcmpkc.

```
sched_ops_t classification_plugin = {
    .policy = SCHED_CLASS_MM,
    .description = "App classification plugin",
    .flags = PMCSCHED_CPUGROUP_LOCK,
    .counter_config = &cconfig,
    .sched_kthread_periodic = sched_kthread_periodic_classification,
    .sched_timer_periodic = sched_timer_periodic_classification,
    .init_plugin = init_plugin_classification,
    .destroy_plugin = destroy_plugin_classification,
    .on_read_plugin = on_read_plugin_classification,
    .on_write_plugin = on_write_plugin_classification,
    .on_active_thread = on_active_thread_classification,
    .on_inactive_thread = on_inactive_thread_classification,
    .on_exit_thread = on_exit_thread_classification,
    .on_new_sample = on_new_sample_classification,
    .on_new_container = on_new_container_classification,
    .on_fork_thread = on_fork_classification};
```

El código muestra las callbacks de PMCSched que este iba a hacer uso, mediante todas estas funciones queremos tener registro de cuando se inicia un contenedor nuevo y sus hilos. Además mediante la implementación de la heurística, también se busca llevar un conteo de las distintas clases de estos hilos para así poder caracterizar la aplicación.

Para poder implementar la heurística se define una estructura en nuestro *plugin* la cual almacena los umbrales necesarios.

```
static struct
{
   unsigned int mpki_threshold;
   unsigned int mpkc_threshold;
   unsigned int stalls_l2_min_threshold;
   unsigned int stalls_l2_max_threshold;
   unsigned int l2_reuse_max_threshold;
   unsigned int l2_reuse_light_sharing_threshold;
   unsigned int diff_threshold;
} classification_config;
```

Estos umbrales pueden ser cambiados sin necesidad de recargar nuestro módulo del kernel gracias a la función *on_write_plugin_classification*. La cual nos permite cambiar sus valores escribiendo el valor del umbral seguido de su nuevo valor en */proc/pmc/sched*. En la función *init_plugin_classification* es donde inicializamos los valores de los distintos umbrales a los definidos en la heurística, además en esta función es donde se activa la configuración de contadores hardware definida en el *plugin*.

Para poder implementar la heurística, debemos implementar los PMCs y definir

las métricas necesarias. Para esto definimos los eventos de los contadores a través del formato *raw* de PMCTrack 2.1. También se define en sus respectivos enums el nombre que se le asigna a los PMCs y métricas de nuestro *plugin*, como podemos observar en este código:

```
static const char *pmcstr_config[] = {
    "pmc0,pmc1,pmc2,pmc3=0x2e,umask3=0x4f,pmc4=0x2e,umask4=0x41,pmc5=0x51,
     um5 0x01,pmc6=0xf1,umask6=0x1f,pmc7=0xa3,umask7=0x05,cm7=0x05",
    NULL};

enum event_indexes
{
    INSTR_EVT = 0,
    CYCLES_EVT,
    UNHALTED_REF_CYCLES_EVT,
    LLC_REFERENCES_EVT,
    LLC_MISSES_EVT,
    L1_REPLACE_EVT,
    L2_LINES_IN_EVT,
    STALLS_L2_MISS_EVT,
    PMC_EVENT_COUNT
};

enum metric_indices
{
    IPC_MET = 0,
    LLCRPKI_MET,
    LLCMPKI_MET,
    LLCRPKC_MET,
    LLCMPKC_MET,
    STALLS_L2_MISS_MET,
    L1REUSE_MET,
    L2REUSE_MET,

    NR_METRICS,
};
```

Para la heurística, hacemos uso de una función auxiliar, la cual será invocada dentro de la callback *on_new_sample_classification*. Dentro de esta callback, se hace uso de la función *check_app_type_status_group*, la cual se encarga de implementar la heurística de clasificación de aplicaciones. Esta hace uso de la estructura de valores umbrales definida anteriormente para decidir la clase de la aplicación. El código de la función es el siguiente:

```
static void check_app_type_status_group(pmcsched_thread_data_t *tp, app_t
     *app, metric_experiment_t *metric_exp, int nr_ways, int cache_usage)
{
     if (metric_exp->metrics[LLCMPKI_MET].count <=
          classification_config.mpki_threshold || metric_exp >
```

```
        metrics[LLCMPKC_MET].count <= classification_config.mpkc_threshold)
    {
        tp->classification.type = CACHE_CLASS_LIGHT;
    }
    else if(metric_exp->metrics[L2REUSE_MET].count >=
        classification_config.l2_reuse_light_sharing_threshold){
        tp->classification.type = CACHE_CLASS_LIGHT;
    }
    else
    {
        if(metric_exp->metrics[STALLS_L2_MISS_MET].count <=
    classification_config .stalls_l2_min_threshold){
            tp->classification.type = CACHE_CLASS_STREAMING;
        }
        else if(metric_exp->metrics[STALLS_L2_MISS_MET].count >=
    classification_config .stalls_l2_max_threshold){
            tp->classification.type = CACHE_CLASS_SENSITIVE;
        }
        else if(metric_exp->metrics[L2REUSE_MET].count >
    classification_config .l2_reuse_max_threshold){
            tp->classification.type = CACHE_CLASS_SENSITIVE;
        }
        else if ((metric_exp->metrics[LLCRPKC_MET].count - metric_exp->
    metrics[LLCMPKC_MET].count) <= classification_config.diff_threshold)
        {
            tp->classification.type = CACHE_CLASS_SENSITIVE;
        }
        else
            tp->classification.type = CACHE_CLASS_STREAMING;
    }
}
```

Con las métricas definidas y nuestra función de clasificación definida, falta guardar el resultado de estas clasificaciones para que luego este pueda ser analizado. Para guardar el resultado, hacemos uso de un array de contadores atómicos de tres posiciones. Los contadores atómicos, clase del kernel Linux *atomic_t*, realizan operaciones que se ejecutan en un ciclo y por tanto no pueden ser interrumpidas. Cada posición es una clase siendo la primera light-sharing, la segunda cache-sensitive y por último streaming. En este array se guardará un registro de la clase de cada hilo clasificado.

```
if (t->classification.type == CACHE_CLASS_LIGHT)
    atomic_inc(&tsk->type_threads[0]);
else if (t->classification.type == CACHE_CLASS_SENSITIVE)
    atomic_inc(&tsk->type_threads[1]);
else if (t->classification.type == CACHE_CLASS_STREAMING)
    atomic_inc(&tsk->type_threads[2]);
```

Por último en la función *sched_timer_periodic_classification*, la cual se llama periódicamente, se realiza la caracterización del contenedor mediante la clase sus hilos.

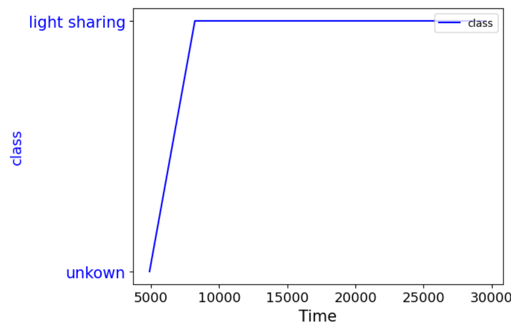
```
static void sched_timer_periodic_classification(void){
   char class[30] = "";
   sched_thread_group_t *cur_group = get_cur_group_sched();
   app_t_pmcsched *cur;
   app_t *app;
   tmon_global_app_t *tsk;

    for (cur=head_sized_list(&cur_group->active_apps); cur!=NULL;
        cur=next_sized_list(&cur_group->active_apps,cur)) {

      app = &cur->app_cache;
      tsk = &cur->sa->tmon_gbl_app;

      if ((atomic_read(&tsk->type_threads[0]) >
          atomic_read(&tsk->type_threads[1])) &&
          (atomic_read(&tsk->type_threads[0]) >
          atomic_read(&tsk->type_threads[2])))
         strcpy(class, "light sharing\n");
      else if ((atomic_read(&tsk->type_threads[1]) >
          atomic_read(&tsk->type_threads[0])) &&
          (atomic_read(&tsk->type_threads[1]) >
          atomic_read(&tsk->type_threads[2])))
      {
         strcpy(class, "cache sensitive\n");
      }
      else if ((atomic_read(&tsk->type_threads[2]) >
          atomic_read(&tsk->type_threads[0])) &&
          (atomic_read(&tsk->type_threads[2]) >
          atomic_read(&tsk->type_threads[1])))
      {
         strcpy(class, "streaming\n");
      }
      else
      {
         strcpy(class, "unkown\n");
      }

      trace_container_class(app->app_cmt_data.rmid, class);

      trace_printk("App %d class is: %s", app->app_cmt_data.rmid, class);
    }
}
```

Para poder obtener los datos de la clasificación realizada por el *plugin* a lo largo del timepo, hacemos uso de la función *trace_container_class* la cual tiene como

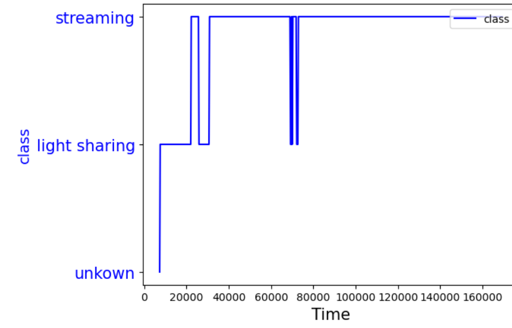
argumentos el RMID y la clase del contenedor. Esta función realmente no es más que una función con código ensamblador vacío, la cual puede ser usada para ver los valores de sus argumentos al ser invocada mediante herramientas como bpftrace o systemtap. Entre estas herramientas, es especialmente interesante systemtap 2, debido a que Het-Harness tiene soporte para scripts de esta, cuya salida se incrusta automáticamente en un log. Es por esto, que para poder obtener los resultados de la clasificación de nuestro *plugin* para previamente analizarlo, fue necesario crear el siguiente script de systemtap:

```
#!/usr/bin/env stap

global time_start

probe begin {
     printf("time,event,class\n");
   time_start=gettimeofday_us()/1000;
}

probe module("mchw_intel_core").function("trace_container_class"){
     relative_time=gettimeofday_us()/1000-time_start;
     printf("%d,%lu,%s\n",relative_time,$rmid,kernel_string($class));
}
```

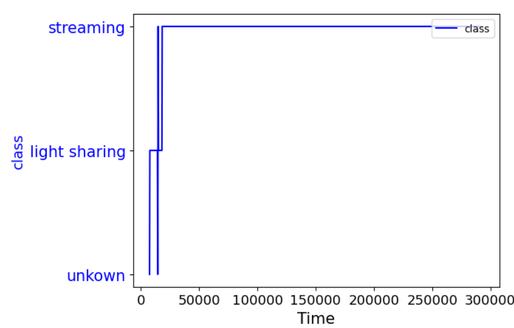
Tras haber implementado y probado en la plataforma Broadwell 5.1.1 los distintos desarrollos realizados para la prueba de la clasificación implementada. Se procede al lanzamiento los experimentos para la caracterización de las aplicaciones cloud. Se decide probar nuestra clasificación sobre estos *benchmarks* cloud debido a disponer de la clase de estos y a haber realizado el ajuste de la heurística sobre los *benchmarks* HPC. Estos experimentos fueron realizados en la plataforma Skylake 5.1.2 otorgando a los programas cloud de 12 cores. Los resultados fueron los siguientes:



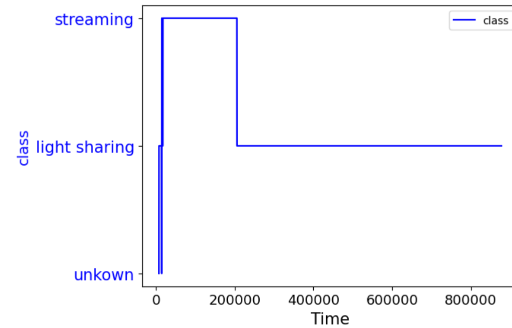
(a) Clasificación In_memory_analitics



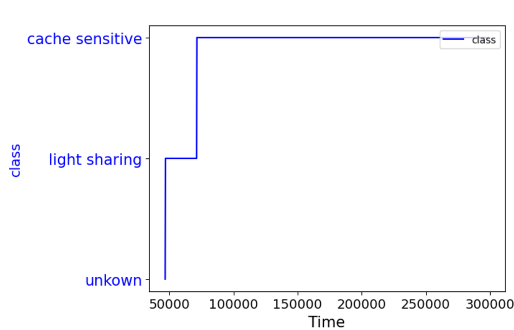
(b) Clasificación graph_analitics PR



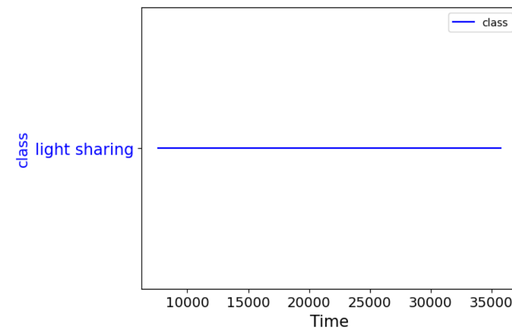
(c) Clasificación graph_analitics CC



(d) Clasificación graph_analitics TC

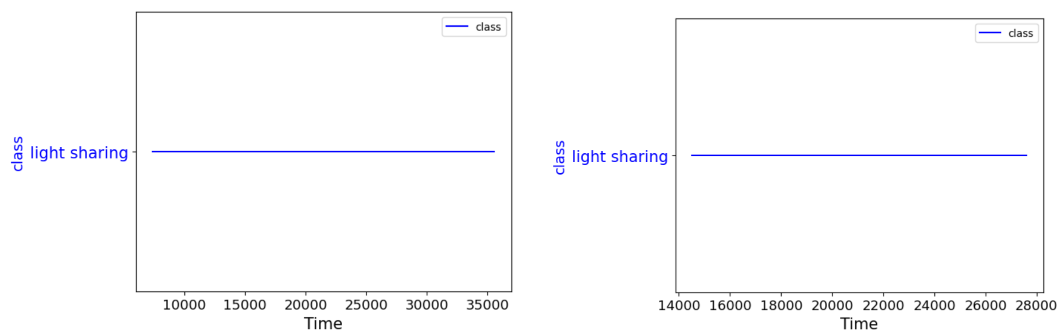


(e) Clasificación data_caching

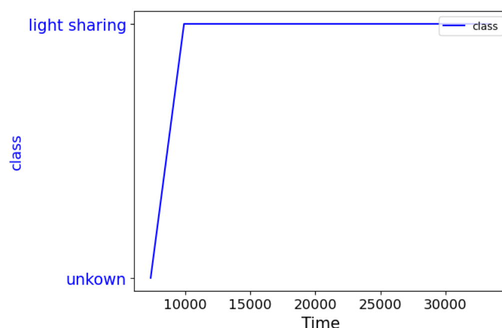


(f) Clasificación data_serving_relational TLP

Figura 6.12: Gráficas de la clasificación mediante métricas 1



(a) Clasificación data_serving_relational OLTP (b) Clasificación data_serving_mysql OLTP



(c) Clasificación data_serving_mysql OLTP

Figura 6.13: Gráficas de la clasificación mediante métricas 2

Como se puede observar en las figuras 6.12 6.13 la heurística de clasificación funcionó bastante bien en lo que respecta a la clasificación de programas light-sharing y streaming aún así ha cometido fallos de predicción con algunas de estas clases. Estos fallos los encontramos en dos de las aplicaciones data_caching y graph_analytics (tc). El primero en anteriores análisis no se veía afectado por el número de vías pero sin embargo la predicción del *plugin* fue cache-sensitive. En el caso de graph_analytics (tc) este ha sido clasificado como cache-sensitive, sin embargo, nuestra predicción lo muestra como streaming al inicio para acabar como light-sharing en la gran mayoría de su ejecución. También nos encontramos con un único caso de caracterización como cache-sensitive 6.12e, este es un ejemplo de una caracterización que no ha sido correcta, ya que como se observa en el experimento realizado del barrido de vías este programa no parece ser afectado ni en ancho de banda, ni en latencia por el número de vías. También encontramos otro ejemplo de caracterización fallida, como es el caso de grahp_analytics tc 6.12d, el cual presenta un comportamiento el cual podríamos clasificar como cache-sensitive o streaming, sin embargo, este ha sido clasificado como light-sharing y en algunos momentos al comienzo como streaming.

Como ya se auguraba al momento de realizar la heurística, esta estaba sujeta a poder mostrar fallos en la caracterización de aplicaciones y así ha sido. Sin embargo, los resultados en la clasificación de programas cloud ha resultado bastante positiva fallando solo 2 clasificaciones de las nueve probadas. Esto sumado el hecho de que el

método de clasificación fue ideado mediante el análisis de programas HPC sorprende su buen comportamiento frente a programas cloud.

Capítulo 7

Conclusiones y Trabajo Futuro

Ejecutar cargas de trabajo de varias aplicaciones de forma simultánea en sistemas multinúcleo, puede llevar a la degradación del rendimiento entre aplicaciones. Esto es debido al fenómeno de *la contención de recursos compartidos* [35, 24]. Esta contención surge debido a los recursos compartidos entre los núcleos de los sistemas CMP (*Chip Multi-Processors*) actuales. Ejemplos de estos recursos compartidos son: la cache de último nivel (LLC), los canales de memoria o los controladores de DRAM. En este trabajo de fin de grado, se ha procedido a analizar el impacto de usar particionado de caché para reducir la degradación en el rendimiento de los programas que se ejecutan simultáneamente en el sistema. Para ello, hemos contemplado mezclas de aplicaciones cloud y HPC.

Para realizar este análisis, hemos tenido en cuenta las clases que se referencian en este artículo [13], las cuales son las siguientes:

- **Light-sharing:** Se tratan de aplicaciones que no se ven afectadas por la contención en la LLC.
- **Cache-sensitive:** Son aquellos programas que sufren mucho cuando se reduce su espacio en la LLC.
- **Streaming:** Son aplicaciones que polucionan la LLC y no se benefician de su espacio en esta.

Con estas clases en cuenta, se ha realizado una clasificación de distintos *benchmarks* HPC y cloud. Esto con el fin de estudiar el uso del particionado de cache para mezclas de aplicaciones cloud y HPC de distintas clases y analizar como afecta en estas el uso del particionado de cache. Adicionalmente, se ha realizado un modelo de predicción de clases de aplicaciones, este ha sido implementado mediante el desarrollo de una extensión del sistema operativo que permite de forma dinámica predecir la clase de los distintos programas.

Son muchas las conclusiones a las que se han llegado durante el transcurso de este trabajo. Quizá la más importante, es el hecho de que tres clases sea insuficiente para la caracterización de programas y haya que actualizar el modelo usado a lo largo de este trabajo. En vista de los resultados de clasificación de aplicaciones, parece un mejor acercamiento cambiar el modelo de clases de aplicaciones que contaba solo con las clases light-sharing, cache-sensitive y streaming por otro que incluya las 5 clases descritas a continuación:

- **Streaming puro:** Se tratan de programas streaming como siempre los hemos entenido. Estos no se ven afectados por el número de vías de cache y consumen muchos recursos de CPU y ancho de banda.
- **Streaming/Cache-sensitive:** Como bien indica su nombre, se trata de una clase intermedia entre cache-sensitive y streaming. Esta presenta características de las dos, teniendo un consumo de CPU y ancho de banda elevados, pero viéndose afectado por su grado de ocupación de la LLC.
- **Light-sharing con workset pequeño:** Esta clase al igual que los light-sharing como siempre se han descrito, apenas consume recursos ni se ve afectado por las vías de cache su rendimiento.
- **Light-sharing con workset mediano:** Necesita de unas pocas vías para tener un buen rendimiento, a partir de 2 a 4 vías este pasa a no verse afectado por estas. Apenas hace uso de la memoria compartida y demanda poco ancho de banda.
- **Cache-sensitive puro:** Se trata del cache-sensitive que entendiamos en nuestra clasificación empleada. Este tiene una alta dependencia del número de vías de cache que dispone con respecto a su rendimiento, no llegan a ser tan demandantes de recursos como los streaming y cuando tienen la cache suficiente baja también su demanda de recursos drásticamente.

La segunda conclusión a la que se ha llegado es que el particionado de caché puede ser una técnica muy efectiva para el aislamiento entre aplicaciones y la gestión de recursos. Sin embargo, puede resultar insuficiente por sí sola. Sin un control adecuado del ancho de banda que demandan las aplicaciones, aquellas cuyo rendimiento depende fuertemente de este recurso pueden sufrir interferencias. Esto se ha evidenciado al mezclar aplicaciones de tipo streaming con aplicaciones sensibles a la caché. En estos casos, ambas demandan un alto ancho de banda, y a pesar de particionar la caché correctamente, la mejora en el rendimiento fue mínima. Esto subraya la importancia de gestionar tanto el ancho de banda como la caché para optimizar el rendimiento en entornos de aplicaciones mixtas.

En cuanto al trabajo futuro, y basándonos en las conclusiones alcanzadas, el camino a seguir es bastante claro. Debido a la nueva forma de clasificación propuesta, se debe realizar una reclasificación de los *benchmarks* de HPC y Cloudsuite. El objetivo es llevar a cabo un análisis exhaustivo del uso de métricas para la clasificación

de programas y desarrollar una nueva heurística para la caracterización de aplicaciones considerando las cinco nuevas categorías. Esto permitirá mejorar la precisión y efectividad en la evaluación y gestión del rendimiento de las aplicaciones.

Además, otra de las posibles líneas de trabajo futuras podría ser el estudio de la gestión del consumo de ancho de banda junto con el particionado de caché para el aislamiento entre aplicaciones. Esto se debe a las conclusiones obtenidas, en las cuales se determina que el particionado de caché no es suficiente sin un adecuado control del ancho de banda. Las plataformas utilizadas durante este trabajo no contaban con soporte para la gestión del ancho de banda. Sin embargo, procesadores más recientes de Intel y AMD permiten el particionado del ancho de banda, lo que permite definir un límite sobre el consumo de ancho de banda de las aplicaciones. Esto puede ayudar a equilibrar la degradación del rendimiento entre aplicaciones. En particular, planteamos el uso de procesadores AMD, que ofrecen mayor granularidad en el particionado del ancho de banda.

Por último, se propone implementar un algoritmo de particionado de caché dinámico utilizando PMCSched, basándose en la nueva función de clasificación que se desarrollará. Los resultados obtenidos a partir del uso de este algoritmo deberán ser evaluados para comprobar su eficacia y mejoras en el rendimiento.

Introduction

"Quality of service or product is not what you put into it.
It's what the customer gets out of it."
— Peter Drucker

Motivation

Multi-core architectures or CMPs (Chip Multi-Processors) are today the predominant design choice for general-purpose systems in multiple industry sectors. In particular, CMPs constitute the de facto architecture for servers in cloud data centers [32, 17, 23]. They are also an essential component of HPC (High-Performance Computing) platforms [33]. Thanks to the latest advances in processor technology and design, many CMP systems now integrate hundreds of cores. Servers based on the AMD EPYC 9654 processor[1], for example, have 192 cores in dual-core configurations.

Making the most of available cores and, more generally, maximizing resource utilization in cloud data centers is critical to reducing costs for cloud providers, and therefore key to increasing their revenue. Achieving this is a significant challenge, and more so given that typical cloud applications often have a highly variable workload throughout the day (e.g., due to fluctuations in the number of customers making use of a web service/portal over time). Note that most of these applications are also latency critical (LC: latency critical) services, such as search engines or many of the artificial intelligence-based services. In the HPC domain, there are many applications that also do not fully utilize the available cores in a CMP system, due to load imbalance between application processes/threads, or the presence of other scalability bottlenecks [14, 33].

To maximize the utilization of CMPs systems, cloud providers typically run multiple services and applications from one or more customers on the same physical server – a practice known as *multi-tenancy*. In addition, to further increase the degree of resource utilization during periods of low server utilization, providers

[1] `https://www.amd.com/es/products/cpu/amd-epyc-9654`

themselves often launch workloads of their own on the same set of servers dedicated to running the workloads of [23, 27, 16] customers. By dynamically managing the resources of the different servers, these vendor workloads are considered non-critical (BE: Best Effort), and therefore their execution can be deferred when it is strictly necessary [34]. In this project, we consider, on the contrary, the use of HPC workloads as a way to increase the degree of utilization of cloud servers in situations of low or moderate demand. The inherent parallelism present in many of these applications makes them particularly suitable for making effective use of multiple idle cores that may be present in the system.

It is worth noting that running different applications simultaneously or concurrently on a multicore system often results in uneven and hard-to-predict performance degradation between applications, due to the phenomenon of shared resource contention [35, 24]. This kind of contention arises from the fact that cores in a current CMP system share critical resources with other neighboring cores, such as a last-level cache (LLC), memory channels or DRAM controllers. Not taking any measures to mitigate the negative effects of competition for these resources between applications often leads to systematic performance degradation and causes unfairness in system usage [13, 25, 28]. Additionally, this may involve non-compliance with service level agreements (SLAs) associated with different cloud applications and services [23, 32]. For example, it has been shown that the effective latency of a cloud service can substantially exceed its allowable value in severe resource contention situations [17, 23].

To enable system software to effectively mitigate the deleterious effects of contention, major processor manufacturers have in recent years opted to include a number of hardware extensions for dynamic shared resource management in their commercial products. In the case of Intel and AMD, these extensions are commercially known as *Intel Resource Director Technology*[2] y *AMD64 Platform QoS Extensions*[3]. Both collections of extensions allow, among other things, to impose per-application quotas on shared resource usage by partitioning the last-level cache or limiting memory bandwidth consumption [26, 30].

Goals

The main objective of this project is to carry out an experimental evaluation to analyze the potential of cache partitioning to provide inter-application isolation in a multi-core server. This analysis has been performed in an environment where the different applications/services run in independent containers on GNU/Linux, since this is a common scenario for deploying applications in the cloud. It should be noted that the analysis of this project aims to lay the foundations for the construction of a shared resource manager implemented in the operating system, and more specifically in the Linux kernel. For this reason, a large part of the experiments performed are

[2]`https://cdrdv2-public.intel.com/789566/356688-intel-rdt-arch-spec.pdf`
[3]`https://developer.amd.com/wp-content/resources/56375.pdf`

based on the use of operating system extensions, some of them implemented as part of this work.

In order to achieve the overall objective of the project, it has been necessary to pursue the following specific objectives:

- **Development of an infrastructure for launching and monitoring containerized applications:** In this project, we opted to use Cloudsuite [1, 19], which consists of a set of benchmarks representative of cloud applications, such as disk-based or in-memory database workloads, as well as massive data processing. In this suite, each benchmark to be executed requires the prior deployment of one or more containers, with components that work cooperatively and require careful data preparation (warmup) and the startup of various key services for their operation.

  Conducting automated experiments with one or more of these benchmarks (some multi-container) required the development of a specific infrastructure to enable the launch and monitoring of workloads, including these applications. For this, the project used the *Het-Harness* tool, used internally in the ArTeCS research group at UCM, as a base. The functionality of this tool has been substantially extended to enable our analysis with containerized applications, both HPC and cloud.

- **Characterization of a representative set of cloud and HPC applications:** Before analyzing the impact of cache partitioning, it was necessary to carry out a characterization of the behavior of a representative set of cloud and HPC applications. In particular, special attention was paid to monitoring the behavior of these benchmarks regarding the use of resources shared between cores (last level cache and memory bandwidth), as well as their degree of CPU utilization over time.

  To extract this type of information –some of which is directly accessible only from within the operating system– the open-source *PMCSched* framework was used. Although this framework already provides support for monitoring LLC usage and bandwidth, it was necessary to extend its functionality to handle the processes of the same container as a whole, which is an essential requirement for our analysis.

- **Analysis of resource contention using mixes of Cloud and HPC applications:** As a result of the findings from the previous objective, a classification of applications into different categories was obtained. This classification leads to the first phase of our analysis of the impact of cache partitioning. For this phase, we propose the construction of a set of mixes formed by cloud and HPC workloads, and we analyze the degradation in performance and/or latency of the different applications or services. Additionally, the impact of statically partitioning the LLC to mitigate the effect of shared resource contention is studied.

- **Development of a dynamic application classification mechanism based on hardware counters:** As the final phase of our analysis, we propose capturing various performance metrics using processor performance monitoring hardware counters [31], to carry out a runtime classification of applications. With this dynamic classification, and the implementation of extensions in the operating system to put it into practice, we aim to lay the foundation for developing automatic LLC partitioning strategies from the operating system itself (a functionality that general-purpose operating systems currently do not offer).

Work Plan

To achieve the specific objectives of the work, the following tasks were carried out, whose timeline is summarized in the diagram in Figure 7.1:

T1 Reading the Cloudsuite documentation and installing the suite on the various servers used.

T2 Familiarizing ourselves with the PMCsched framework and deploying it on the various multicore platforms used in the project.

T3 Completing various introductory tutorials on the Het-Harness tool and analyzing the existing proprietary documentation on this tool.

T4 Developing adaptations of Cloudsuite to meet the needs of our project.

T5 Updating the Het-Harness and PMCSched tools to adapt to the needs of our project.

T6 Analyzing metrics for application characterization. Creating heuristics for application characterization.

T7 Implementing the classification method in PMCSched.

T8 Conducting characterization experiment 1: Running the Cloudsuite applications. Creating graphs and analyzing the generated graphs.

T9 Conducting characterization experiment 2: Running all applications, both cloud and HPC, limiting their shared resource (last level cache). Creating graphs and analysis.

T10 Conducting characterization experiment 3: Mixing cloud and HPC applications. Creating graphs and analyzing the generated graphs.

T11 Writing the report.

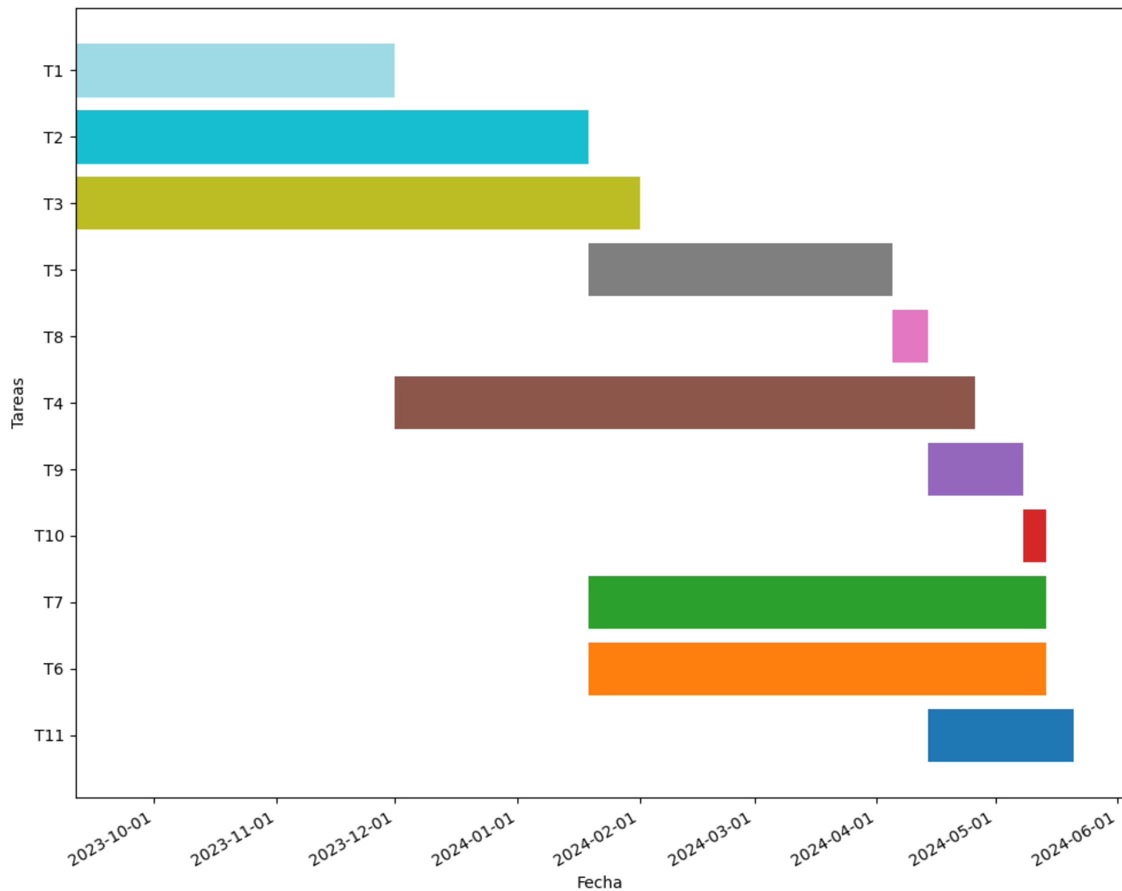


Figure 7.1: Gantt Chart

Structure of the report

The rest of this document is structured as follows:

- Chapter 2 describes the various tools used during the course of this work. This description includes the layers that compose them and their usage.
- Chapter **??** details the changes made to the Het-Harness tool to enable its use with containers and cloud applications.
- Chapter 4 outlines the modifications made to the PMCSched framework for better support in the use of containers.
- Chapter 5 provides a description of the experimental platforms used, as well as the various experiments conducted and their development. For each experiment performed, the results are analyzed and conclusions are drawn.
- Chapter 6 details the steps followed to develop a heuristic for application classification and the creation of an operating system extension that incorporates this heuristic for real-time application classification. Additionally, this chapter

includes the analysis of the results after classifying cloud application benchmarks using this plugin.

- Chapter 7 lists the final conclusions reached throughout this bachelor's thesis. This chapter also outlines potential future work directions.

Conclusions and Future Work

Running multiple application workloads simultaneously on multi-core systems can lead to performance degradation due to the phenomenon of *shared resource contention* [35, 24]. This contention arises from shared resources among the cores of current CMP (*Chip Multi-Processors*) systems, such as the last-level cache (LLC), memory channels, or DRAM controllers. This project analyzed the impact of using cache partitioning to reduce performance degradation in programs running simultaneously on the system, focusing on a mix of cloud and HPC applications.

For this analysis, we considered the classes referenced in this article [13], which are as follows:

- **Light-sharing:** Applications that are not affected by LLC contention.
- **Cache-sensitive:** Programs that suffer significantly when their LLC space is reduced.
- **Streaming:** Applications that pollute the LLC and do not benefit from its space.

Taking these classes into account, we classified various HPC and cloud *benchmarks* to study the use of cache partitioning for mixed cloud and HPC applications and analyze its impact. Additionally, we developed a predictive model for application classes, implemented through an operating system extension that dynamically predicts the class of various programs.

Several conclusions were reached during this work. Perhaps the most significant is that three classes are insufficient for characterizing programs, necessitating an update to the model used throughout this thesis. Based on the application classification results, it seems more effective to switch from the model with only light-sharing, cache-sensitive, and streaming classes to one that includes the following five classes:

- **Pure Streaming:** Programs that, as traditionally understood, are not affected by the number of cache ways and consume significant CPU and bandwidth resources.

- **Streaming/Cache-sensitive:** An intermediate class between cache-sensitive and streaming, with high CPU and bandwidth consumption but affected by LLC occupancy.
- **Light-sharing with small workset:** Similar to traditional light-sharing applications, these consume minimal resources and are not affected by cache ways.
- **Light-sharing with medium workset:** Require a few cache ways (2-4) for good performance and minimally use shared memory and bandwidth.
- **Pure Cache-sensitive:** Traditional cache-sensitive programs highly dependent on the number of cache ways for performance, but less demanding than streaming programs and with reduced resource demands when sufficient cache is available.

The second conclusion is that cache partitioning can be very effective for isolating applications and managing resources, but it may be insufficient on its own. Without proper bandwidth control, applications that heavily depend on this resource can experience interference. This was evident when mixing streaming applications with cache-sensitive ones. Both types demand high bandwidth, and even with proper cache partitioning, performance improvement was minimal. This highlights the importance of managing both bandwidth and cache to optimize performance in mixed application environments.

Regarding future work, based on the conclusions reached, the path forward is clear. Due to the newly proposed classification model, a reclassification of HPC and Cloudsuite benchmarks is necessary. The goal is to conduct an exhaustive analysis of metrics for program classification and develop a new heuristic for application characterization considering the five new categories. This will enhance the precision and effectiveness of application performance evaluation and management.

Additionally, another possible future research direction is the study of bandwidth consumption management alongside cache partitioning for application isolation. This is due to the conclusions that cache partitioning alone is insufficient without proper bandwidth control. The platforms used in this work did not support bandwidth management. However, more recent Intel and AMD processors allow bandwidth partitioning, enabling the definition of bandwidth consumption limits for applications. This can help balance performance degradation among applications. We propose using AMD processors, which offer greater granularity in bandwidth partitioning.

Finally, we propose implementing a dynamic cache partitioning algorithm using PMCSched, based on the new classification function to be developed. The results from using this algorithm should be evaluated to assess its effectiveness and performance improvements.

Contribuciones Personales

En esta sección indicaremos las contribuciones de cada participante al proyecto.

Diego Pellicer

Diego se ha encargado de varias de las tareas de este TFG. Empezando por la parte de aprender las nuevas herramientas que han sido utilizadas, este se ha encargado de entender y aprender el uso de PMCSched. Esto se ha logrado a través de las distintas reuniones con los directores. Además en reuniones se explicó el funcionamiento de Het-Harness el cual también aprendido y entendido.

Adicionalmente en estas reuniones, se nos puso a nuestra disposición una serie de artículos relacionados con el tema que se trata en este trabajo de fin de grado. La primera lectura de artículos que realiza es aquella sobre el los problemas al ofrecer servicios cloud y lanzar otro tipos de aplicaciones. A estas lecturas, se le agregan lecturas sobre la contención de recursos y métodos para garantizar el aislamiento de aplicaciones.

Siguiendo con los contenidos consumidos para un mejor acercamiento al objetivo e inicio del trabajo del TFG, se realizó una reunión para explicar las tecnologías del particionado de cache. En esta reunión se nos dió contexto de las distintas formas de particionado de cache existentes y de las usadas por AMD e Intel. Expecialmente se realiza una lectura para mayor entendimiento de la tecnología Intel CAT [5] ya que se va a trabajar con procesadores Intel a lo largo de este trabajo.

Tras haber realizado las lecturas y el acercamiento previo necesario para empezar a trabajar de lleno en el TFG, Diego realiza un primer *plugin* de PMCSched. Este *plugin* se encargaría de sacar la información de 8 PMCs distintos y con estos generar 7 métricas. Diego se encargo de generar el nuevo *plugin*, sacando el código de los PMCs en formato *raw* como se cuenta en la sección 2.1 de estos mediante la página que nos brinda Intel [3]. Este *plugin* se trataba de uno muy básico el cual mediante la *callback on_new_sample* actualizaba el valor de los contadores definidos y sus métricas. Luego también se encargó de realizar las consecuentes pruebas del *plugin*

a través de *benchmarks* HPC en la herramienta Het-Harness, la cual se detalla en la sección 2.2.

Una vez el *plugin* se encontraba terminada y funcionando a la perfección, se realizó una nueva reunión. Esta con el fin de ver cual era lo siguiente por hacer. En esta se llegó a la conclusión de que la siguiente tarea, sería crear un script de Het-Harness para el barrido de vías de distintos *benchmarks* HPC. Este script además del barrido de vías hace uso de PMCTrack para sacar las métricas necesarias para más adelante realizar un análisis del valor de estas para los distintos *benchmarks*.

El script, una vez terminado, fue lanzado por Diego en nuestra plataforma Broadwell definida en la sección 5.1.1. Esta ejecución dio como resultado varios 20 logs por cada *benchmark* HPC. Todos estos logs se convirtieron mediante un script en python en un csv el cual más tarde sería analizado.

Con el csv preparado, Diego procede a realizar el análisis de los resultados vía un Jupyter Notebook. Este análisis realizado, está explicado a mayor detalle dentro del capítulo 6. En este Diego se encargó al completo de todo el análisis, comenzando por crear una matriz de correlación para reducir el número de métricas a analizar. Una vez hecha la matriz, llegó a la conclusión de que un árbol de decisión era una buena idea para obtener las métricas que mayor entropía tenía, y que por tanto más agrupaban los datos. Se realizaron dos árboles, uno completo y otro de altura 4, se analizaron los resultados y se sacaron varias métricas candidatas. También se encargó de hacer uso del algoritmo de *clusthering* de k-means y analizar los resultados obtenidos.

Con estos dos algoritmos aplicados, se realiza un análisis de cada métrica con el fin de encontrar valores umbrales. Para esto se mira para cada número de vías la media, mínimo y máximo de cada métrica en cada clase. Con este análisis se concluye una heurística para clasificar entre las distintas clases.

Luego de esto se hicieron varias reuniones más en las cuales se acordó realizar un análisis de un conjunto de *benchmarks* HPC y cloud, esto con el fin de clasificarlos. Diego se encargó de junto a Yikang realizar el análisis de las distintas gráficas obtenidas, además junto con los directores del TFG se valoró los resultados obtenidos llegando a distintas conclusiones bastante interesantes.

Tras haber clasificado los distintos programas, quedaba por hacer, el análisis de como afectaba el particionado de cache a distintas cargas de programa. En conjunto a su compañero de TFG, se eligió las distintas mezclas de aplicaciones que se iban a realizar y las vías asignadas a cada programa para cada mezcla. Una vez obtenido los resultados, Diego formó parte de análisis de las gráficas que mostraban los resultados obtenidos y aportó en las distintas conclusiones y vías de trabajo futuro que estas aportaban.

Por último, Diego realizó un *plugin* de PMCSched el cual implementaba la heurística previamente definida. Una vez esta fue desarrollada, se preparó y lanzó un script de Het-Harness para obtener la predicción de clases de este *plugin* para nues-

tros *benchmarks* cloud. También realizó las gráficas que muestran los resultados de este experimento y analizó los resultados y las conclusiones que este análisis arrojaba.

En la redacción de la memoria de este TFG, Yikang se ha encargado de las siguientes secciones:

- El capítulo 1 en conjunto a Yikang y los directores, con algunas partes en solitario como la estructura de la memoria.
- La sección de PMCSched dentro del capítulo 2.
- El capítulo 4 en su totalidad.
- Partes del capítulo 5.
- La redacción de capítulo 6.
- Y por último, la redacción de las conclusiones finales (capítulo 7), recibiendo ayuda en esta por parte de Yikang.

Yikang Chen

Yikang se ha ocupado de la parte de Cloudsuite, aprender a usar Docker y familiarizarse con los *benchmarks* de Cloudsuite. En la etapa inicial del proyecto, Yikang comenzó con la instalación de las herramientas necesarias para completar algunos tutoriales de Docker en su propio equipo y en la plataforma de Broadwell (plataforma explicada en la sección 5.1.1).

Después de adquirir conocimientos sobre el uso de Docker y las características de los *benchmarks* de Cloudsuite, se llevaron a cabo reuniones con el director para explicar la estructura y funcionamiento de la herramienta Het-Harness. En reuniones posteriores, también se detallaron las modificaciones que se han hecho a la herramienta para satisfacer las necesidades que presentaba nuestro proyecto (las modificaciones se explican en el capítulo 3).

Antes de iniciar los experimentos de clasificación de los *benchmarks*, Yikang se encargó de crear archivos Markdown que recopilaban el funcionamiento de los distintos *benchmarks* de Cloudsuite. Estos documentos detallaban cada paso necesario, desde la creación de las imágenes y contenedores requeridos, hasta la ejecución de los *benchmarks* y las salidas que generaban. Se incluían instrucciones precisas sobre cada comando necesario, proporcionando una guía completa y detallada para asegurar una ejecución correcta y eficiente de los *benchmarks*.

En la fase inicial de los experimentos de clasificación (ver sección 5.4), Yikang se encargó de lanzar una parte de los *benchmarks* de Cloudsuite en la plataforma Skylake (plataforma descrita en la sección 5.1.2) y de crear las gráficas lineales

con las salidas de las ejecuciones de los *benchmarks* necesarios en esta fase experimental, para luego analizarlas con los participantes del TFG. Con el objetivo de transformar los datos de salida de las ejecuciones de los *benchmarks* a ficheros CSV con los parámetros y características que nos interesaban, se desarrollaron scripts en Python. Estos scripts, ubicados en el mismo directorio que los ficheros log (los cuales contienen los datos de salida de los *benchmarks*), fueron creados para cumplir esta finalidad.

Para consolidar todas las gráficas creadas durante las fases experimentales, se utilizó Jupyter Notebook. En este entorno, Yikang creó y agrupó todas las gráficas generadas a partir de las salidas de los distintos experimentos. Además, desarrolló funciones globales para asegurar que el *notebook* fuera mantenible y legible.

En la segunda fase de los experimentos, ejecutamos exhaustivamente todos los *benchmarks* previstos, tanto los de aplicaciones cloud como los de HPC. Yikang se encargó de desarrollar scripts en Python para *parsear* las salidas de estas ejecuciones, automatizando la extracción de datos relevantes.

En la tercera y última fase de los experimentos, todos los participantes estuvieron activamente involucrados en la ejecución de los *benchmarks* necesarios en la plataforma Skylake, utilizando la herramienta Het-Harness. Se realizaron reuniones para decidir qué *benchmarks* seleccionar para esta fase experimental, tras lo cual se crearon los scripts en Het-Harness para ejecutar los *benchmarks* elegidos.

Yikang se encargó de desarrollar los scripts para *parsear* las salidas de los *benchmarks* y generar las gráficas correspondientes. Posteriormente, estas gráficas fueron analizadas en detalle durante las reuniones con el director, lo que permitió una evaluación exhaustiva de los resultados obtenidos y la efectividad de las técnicas implementadas.

En la redacción de la memoria de este TFG, Yikang se ha encargado de las siguientes secciones:

- El capítulo 1 en conjunto a Diego y los directores, con algunas partes en solitario como el plan de trabajo.
- Las secciones de Cloudsuite y Het-Harness del capítulo 2.
- Redacción del capítulo 3.
- Partes del capítulo 5.
- Modificaciones del capítulo 7.
- Y por último, las traducciones al inglés de la introducción, resumen y conclusión.

Bibliografía

[1] Cloudsuite, a benchmark suite. Disponible en https://www.cloudsuite.ch/.

[2] Documentación de in-memory analitics. Disponible en https://github.com/parsa-epfl/cloudsuite/blob/main/docs/benchmarks/in-memory-analytics.md.

[3] Eventos de contadores para arquitecturas intel. Disponible en https://perfmon-events.intel.com.

[4] Grupo artecs. Disponible en https://artecs.dacya.ucm.es/.

[5] Introduction to cache allocation technology in the intel® xeon® processor e5 v4 family. Disponible en https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html.

[6] Página web de memcached. Disponible en https://memcached.org/.

[7] Página web de spark. Disponible en https://spark.apache.org/.

[8] Repositorio de cloudsuite. Disponible en https://github.com/parsa-epfl/cloudsuite/tree/main/benchmarks/data-analytics/latest.

[9] Al-Dhuraibi, Y., Paraiso, F., Djarallah, N. y Merle, P. Elasticity in cloud computing: State of the art and research challenges. *IEEE Transactions on Services Computing*, vol. 11(2), páginas 430–447, 2018.

[10] Alexandros-Herodotos HARITATOS, Nikela PAPADOPOULOU, Konstantinos NIKAS y Georgios GOUMAS and Nectarios KOZIRIS. Contention-aware scheduling policies for fairness and throughput. 2016.

[11] Bailey, D. H., Barszcz, E. y Barton et al., J. T. The NAS parallel benchmarks—summary and preliminary results. En *Supercomputing '91*, páginas 158–165. 1991.

[12] Bienia, C. et al. The PARSEC Benchmark Suite: Characterization and Architectural Implications. En *Proc. of PACT'08*. 2008.

[13] Bilbao, C., Saez, J. C. y Prieto-Matias, M. Divide&content: A fair os-level resource manager for contention balancing on numa multicores. *IEEE Transactions on Parallel and Distributed Systems*, vol. 34(11), páginas 2928–2945, 2023.

[14] Bilbao, C., Saez, J. C. y Prieto-Matias, M. Flexible system software scheduling for asymmetric multicore systems with PMCSched: A case for Intel Alder Lake. *Concurrency and Computation: Practice and Experience*, vol. 35(25), 2023.

[15] Che, S. et al. Rodinia: A benchmark suite for heterogeneous computing. En *Proc of IISWC '09*, páginas 44–54. 2009.

[16] Chen, R., Wu, J., Shi, H., Li, Y., Liu, X. y Wang, G. DRLPart: A Deep Reinforcement Learning Framework for Optimally Efficient and Robust Resource Partitioning on Commodity Servers. En *Proc. of HPDC '21*. 2021. ISBN 9781450382175.

[17] Chen, S., Delimitrou, C., Martinez, J. F. y Others, F. Parties: Qos-aware resource partitioning for multiple interactive services. En *Proc. of ASPLOS*. 2019. ISBN 9781450362405.

[18] Chitsaz, H. et al. A partition function algorithm for interacting nucleic acid strands. *Bioinformatics*, vol. 25(12), páginas i365–i373, 2009.

[19] Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A. D., Ailamaki, A. y Falsafi, B. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. En *Proc. of ASPLOS '12*, ASPLOS XVII, página 37–48. Association for Computing Machinery, New York, NY, USA, 2012. ISBN 9781450307598.

[20] Garcia-Garcia, A., Saez, J. C. y Prieto-Matias, M. PBBCache: an open-source parallel simulator for rapid prototyping and evaluation of cache-partitioning and cache-clustering policies. *J. Computat. Science*, página 101102, 2020. ISSN 1877-7503.

[21] Garcia-Garcia, A. et al. LFOC: A lightweight fairness-oriented cache clustering policy for commodity multicores. En *Proc. of ICPP'19*, páginas 14:1–14:10. 2019.

[22] Hourd, J., Fan, C., Zeng, J., Zhang, Q., Best, M. J., Fedorova, A. y Mustard, C. Exploring practical benefits of asymmetric multicore processors. Disponible en `https://people.ece.ubc.ca/sasha/papers/PESPMA-09.pdf`.

[23] Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P. y Kozyrakis, C. Heracles: improving resource efficiency at scale. En *Proc. of ISCA '15*, páginas 450–462. 2015.

[24] Mittal, S. A survey of techniques for cache partitioning in multicore processors. *ACM Comput. Surv.*, vol. 50(2), páginas 27:1–39, 2017. ISSN 0360-0300.

[25] Nikas, K., Papadopoulou, N., Giantsidi, D., Karakostas, V., Goumas, G. y Koziris, N. DICER: Diligent cache partitioning for efficient workload consolidation. En *Proc. of ICPP '19*. 2019. ISBN 9781450362955.

[26] Park, J., Park, S. y Baek, W. Copart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers. En *Proc. of EuroSys '19*. 2019. ISBN 9781450362818.

[27] Patel, T. y Tiwari, D. Clite: Efficient and QoS-aware co-location of multiple latency-critical jobs for warehouse scale computers. En *Proc. of HPCA '20*, páginas 193–206. 2020.

[28] Roy, R. B., Patel, T. y Tiwari, D. Satori: Efficient and fair resource partitioning by sacrificing short-term benefits for long-term gains. En *Proc. of ISCA '21*, páginas 292–305. 2021.

[29] Rubio, J., Bilbao, C., Saez, J. C. y Prieto-Matias, M. Exploiting elasticity via os-runtime cooperation to improve cpu utilization in multicore systems. En *2024 32nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, páginas 35–43. 2024.

[30] Saez, J. C., Castro, F., Fanizzi, G. y Prieto-Matias, M. LFOC+: A Fair OS-Level Cache-Clustering Policy for Commodity Multicore Systems. *IEEE Transactions on Computers*, vol. 71(8), páginas 1952–1967, 2022.

[31] Saez, J. C., Pousa, A., Rodriguez, R., Castro, F. y Prieto-Matias, M. PMCTrack: Delivering performance monitoring counter support to the OS scheduler. *The Computer Journal*, vol. 60(1), páginas 60–85, 2017.

[32] Shahrad, M., Elnikety, S. y Bianchini, R. Provisioning differentiated last-level cache allocations to VMs in public clouds. En *Proc. of SoCC '21*, página 319–334. 2021. ISBN 9781450386388.

[33] Zacarias, F. V., Petrucci, V., Nishtala, R., Carpenter, P. y Mossé, D. Intelligent colocation of HPC workloads. *Journal of Par. and Distrib. Computing*, vol. 151, 2021. ISSN 0743-7315.

[34] Zhao, L., Cui, Y., Yang, Y., Zhou, X., Qiu, T., Li, K. y Bao, Y. Component-distinguishable co-location and resource reclamation for high-throughput computing. *ACM Trans. Comput. Syst.*, vol. 42(1–2), 2024. ISSN 0734-2071.

[35] Zhuravlev, S. et al. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv.*, vol. 45(1), páginas 4:1–4:28, 2012. ISSN 0360-0300.