Análisis del impacto del particionado de caché en el rendimiento de las aplicaciones multihilo
Analyzing the impact of cache-partitioning in the performance of multithreaded applications

Trabajo de Fin de Grado
Curso 2023–2024

Autor
Daniel Martín Gómez

Director
Juan Carlos Sáez Alcaide

Doble Grado en Ingeniería Informática y Matemáticas
Facultad de Informática
Universidad Complutense de Madrid

Análisis del impacto del particionado de caché en el rendimiento de las aplicaciones multihilo
Analyzing the impact of cache-partitioning in the performance of multithreaded applications

Trabajo de Fin de Grado en Ingeniería Informática

Autor
Daniel Martín Gómez

Director
Juan Carlos Sáez Alcaide

Convocatoria: *Septiembre 2024*

Doble Grado en Ingeniería Informática y Matemáticas
Facultad de Informática
Universidad Complutense de Madrid

13 de Septiembre de 2024

Agradecimientos

Quiero expresar mi agradecimiento a mi director de TFG, Juan Carlos, no sólo por la orientación y dedicación en lo que es el trabajo en sí, sino por su ayuda y comprensión con ciertas circunstancias personales que han hecho que este trabajo se haya alargado más de dos años (con cambio de tema de por medio).

También a familia y amigos, sin los que no habría sido posible llegar hasta aquí. Son muchas las amistades que me llevo de estos seis años en la UCM.

Resumen

Análisis del impacto del particionado de caché en el rendimiento de las aplicaciones multihilo

Los procesadores multinúcleo son actualmente la arquitectura dominante de propósito general. Cabe destacar que en este tipo de arquitecturas los distintos núcleos (*cores*) que integra el procesador no son procesadores totalmente independientes, ya que típicamente existen recursos compartidos entre *cores*, como ciertos niveles de caché (L2 o L3), los canales de acceso a memoria principal o el controlador de DRAM. El problema de tener estos recursos compartidos es que las aplicaciones que se ejecutan simultáneamente en el sistema compiten por el uso de los mismos, lo cual puede afectar al rendimiento de estas aplicaciones de forma desigual, provocando injusticia (*unfairness*) en el acceso a estos recursos, así como la degradación global del rendimiento del sistema.

Una medida para mitigar los efectos negativos de la contención es el particionado de las cachés compartidas. Recientemente los principales fabricantes de hardware han introducido en sus procesadores comerciales (típicamente de gama alta), soporte *hardware* para particionar la caché desde el *software* del sistema. Hay numerosas propuestas de investigación que utilizan este soporte *hardware* de particionado de caché para garantizar aislamiento entre distintas aplicaciones, asignando distintas particiones a distintas aplicaciones. No obstante, en estas estrategias todos los hilos de una misma aplicación (multihilo) comparten la misma partición.

En este trabajo se ha explorado el potencial derivado de particionar la caché entre los distintos hilos de una aplicación, en un escenario en el que la aplicación se ejecuta sola en el sistema. Para llevar a cabo nuestro análisis se ha implementado soporte en el kernel Linux para realizar particionado de caché automático (guiado por el sistema operativo) en dos plataformas distintas, una que soporta particionado para la caché L2 y otra para la caché L3. El análisis experimental se ha realizado utilizando un amplio rango de aplicaciones —principalmente HPC (*High Performance Computing*) — en dos procesadores de Intel, uno de ellos con microarquitectura Alder Lake y otro de la familia Intel Broadwell-EP.

Palabras clave

Kernel Linux, HPC, Particionado de caché, Intel CAT, Contención de recursos, PMCSched, PMTrack, Intel Alder Lake, Intel Broadwell-EP.

Abstract

Analyzing the impact of cache-partitioning in the performance of multithreaded applications

Multicore processors constitute the main architecture choice for general purpose systems. Its worth noting that the different cores on a processor are not truly independent one from each other, as there are usually shared resources between cores, like some levels of cache (L2 or L3), memory access channels or the DRAM controller. The contention that naturally appears when multiple applications compete for the use of shared resources among cores may lead to substantial performance degradation, and have a negative impact on fairness. Moreover, the effect on contention is unevenly distributed between applications.

Partitioning shared caches has been proven effective to mitigate shared-resource contention effects. Lately, the main chip manufacturers (Intel and AMD) have added cache partitioning hardware support to their commercial processors. Several research articles have been published on cache partitioning, using this hardware support to effectively achieve isolation between applications by allocating different partitions to different applications. However, in these strategies all threads of a multithreaded application are assigned to the same partition.

In this work we have evaluated the potential benefit of cache partitioning at the thread level, when a multithreaded application runs alone on the system. To this end we have implemented tools for cache partitioning support on top of PMCSched, which integrates it into the Linux kernel, for two different platforms: One with L2 partitioning support and another with L3 partitioning support. Experiments and subsequent analysis have been conducted on these platforms using a wide range of applications, mainly HPC (High Performance Computing) benchmarks.

Keywords

Linux kernel, HPC, Cache partitioning, Intel CAT, Resource contention, PMCSched, PMCTrack, Intel Alder Lake, Intel Broadwell-EP.

Índice

Índice de figuras

Capítulo 1

Introducción

Los sistemas multinúcleo (CMPs) son una de las arquitecturas más comunes de propósito general hoy en día. En esta clase de sistemas pueden producirse problemas de rendimiento derivados de la competencia que diferentes aplicaciones hacen por los recursos del sistema que están compartidos entre núcleos (pues los núcleos no son realmente procesadores independientes) [5]. Este es el llamado *problema de la contención de recursos compartidos* [6] [7], que puede provocar problemas de rendimiento desiguales entre aplicaciones y difíciles de predecir, puesto que dependen de cuáles sean las aplicaciones que estén ejecutándose simultáneamente, y del punto de ejecución en el que se encuentren [8] [9].

La memoria caché es uno de esos recursos compartidos. Si bien se han venido proponiendo soluciones a este problema desde hace décadas [10] no ha sido hasta recientemente que se ha añadido finalmente el soporte *hardware* necesario para particionar la caché en procesadores comerciales (tanto de Intel como de AMD) [11]. Desde la inclusión de este soporte *hardware* se han ido proponiendo diversas soluciones creadas específicamente con tipos específicos de *hardware* de particionado como objetivo [5]. Debido a que el *hardware* disponible permite típicamente particionado a nivel de vía (es decir, se asignan vías a particiones) y a que usualmente el número de vías de una caché es muy limitado, muchas opciones implementan estrategias de *clustering* en lugar de particionado estricto [12] [13]. Estas técnicas de *clustering* consisten en que aplicaciones distintas pueden compartir una partición de caché. Por ejemplo, una estrategia de *clustering* puede consistir en que todas las aplicaciones que no hacen un acceso eficiente a la caché (su rendimiento no mejora cuando disponen de mucho espacio de caché, ni empeora si reciben un espacio muy reducido) compartan una partición pequeña de la caché. De esta forma queda más espacio de la caché disponible para asignar a aquellas aplicaciones que sí muestren degradación de rendimiento cuando se ejecutan en una caché compartida. Por otro lado, las técnicas de particionado estricto consisten en que cada aplicación recibe una partición que las aísla completamente del resto de aplicaciones (no comparten la partición con otras aplicaciones). En general, en las propuestas de investigación se utiliza el particionado de caché (*clustering* o estricto) para intentar garantizar aislamiento de cada aplicación con respecto al resto [12] [13] [14] [15] [16] [10]. Por

tanto, todos los hilos de una aplicación estan asignados a la misma particion de caché. De esta forma se busca solucionar la contención de recursos entre distintas aplicaciones que se ejecutan simultáneamente en un sistema.

1.1. Objetivos

El objetivo de este trabajo es evaluar los beneficios asociados a particionar la caché entre los distintos hilos de una misma aplicación, cuando esta se ejecuta en solitario en el sistema. En caso de detectar que esta aproximación sea prometedora se pretende también desarrollar políticas de gestión de la memoria caché que permitan mejorar el rendimiento de aplicaciones multihilo o en contenedores (potencialmente multiproceso).

Este objetivo surge de la idea intuitiva de que, en algunas aplicaciones multihilo con caché compartida, algunos hilos podrían reemplazar líneas de caché que otros hilos estén reutilizando, creando así una cierta polución de caché que podría llegar a afectar al rendimiento.

Para desarrollar este objetivo ha sido necesario completar otros objetivos menores, que detallamos a continuación:

- **Implementar soporte para gestionar particionado de caché por hilo en el kernel Linux**: Desarrollamos *plugins* para PMCSched (un *software* de prototipado rápido de políticas de planificación y gestión de recursos para el kernel Linux) para implementar las políticas de particionado usando Intel CAT (*Cache Allocation Technology*) como soporte *hardware* para el particionado en distintas arquitecturas. Para ello fue necesario desarrollar dos *plugins*: Uno para particionar la caché L2 y otro para la caché L3.

- **Analizar los beneficios derivados de particionar la caché a nivel de hilo**: Utilizar el soporte desarrollado en el punto anterior para analizar los beneficios de particionar la caché entre los distintos hilos de una misma aplicación (que se ejecute en solitario en el sistema). Para llevar a cabo esto seleccionamos una serie de *benchmarks* HPC (*High Performance Computing*) populares y evaluamos las diferencias en rendimiento al ejecutarlos con la caché compartida y con la caché particionada de forma equitativa entre todos los hilos (particionado estático). Para dichas evaluaciones utilizamos Het-Harness, un *framework* que permite realizar experimentos para evaluar políticas de gestión de recursos. También tuvimos que añadir la *suite* PBBS [17] a Het-Harness. La evaluación se llevó a cabo en dos plataformas experimentales. La primera es una máquina de escritorio con procesador Intel Core i9-12900K y microarquitectura Alder Lake, que dispone de soporte para particionado de caché L2. La segunda tiene un procesador Intel Xeon CPU E5-2620 v4 con microarquitectura Broadwell y particionado de caché L3. A partir de ahora nos referiremos a estas máquinas por su microarquitectura, como plataforma Alder Lake y plataforma Broadwell-EP respectivamente (para hacer más ágil la redacción).

- **Obtención de muestreo de contadores y análisis**: En esta fase extrajimos información de contadores *hardware* de las aplicaciones estudiadas utilizando

PMCTrack [18], calculamos métricas y procedimos a hacer un análisis de los datos obtenidos para tratar de explicar los resultados de la política de particionado estático, y para tratar de detectar situaciones favorables al particionado.

1.2. Plan de trabajo

Este trabajo se ha desarrollado a base de tutorías a intervalos irregulares en las que se discutían las siguientes tareas a realizar, se discutían resultados o resolvían dudas o problemas surgidos durante el desarrollo del TFG por parte del alumno. Además, el tutor estaba disponible para resolver dudas puntuales a través de una aplicación de mensajería instantánea y correo electrónico. La frecuencia de las tutorías dependía de las características en cuestión de las tareas que estuviesen siendo realizadas en ese momento. Por la cantidad de dependencias entre unas tareas y otras ha sido un trabajo muy lineal, habiéndose realizado a grandes rasgos las siguientes tareas, cuya planificación temporal se muestra en la figura 1.1:

T1 Familiarizarse con la investigación realizada respecto a particionado de caché (leyendo algunos artículos como [12], [13] o [10]).

T2 Familiarizarse con PMCTrack y PMCSched.

T3 Familiarizarse con Het-Harness mediante lectura de la documentación y seminario introductorio.

T4 Familiarizarse con Intel RDT e Intel CAT (mediante lectura del manual).

T5 Implementación de la API de particionado de caché L2 en PMCTrack.

T6 Implementación del plugin de particionado de caché L2 en PMCSched.

T7 Realización de experimentos de particionado estático en Alder Lake: Todos salvo *suite* PBBS.

T8 Añadir soporte para la suite PBBS en Het-Harness.

T9 Realización de experimentos adicionales de particionado estático en Alder Lake: *suite* PBBS.

T10 Análisis de datos.

T11 Implementación del plugin de particionado de caché L3 en PMCSched.

T12 Realización de experimentos de particionado estático en Broadwell-EP.

T13 Obtención de medidas de contadores hardware de los *benchmarks*.

T14 Realización de experimentos adicionales de particionado estático en Broadwell-EP: contenedores.

T15 Redacción de la memoria.

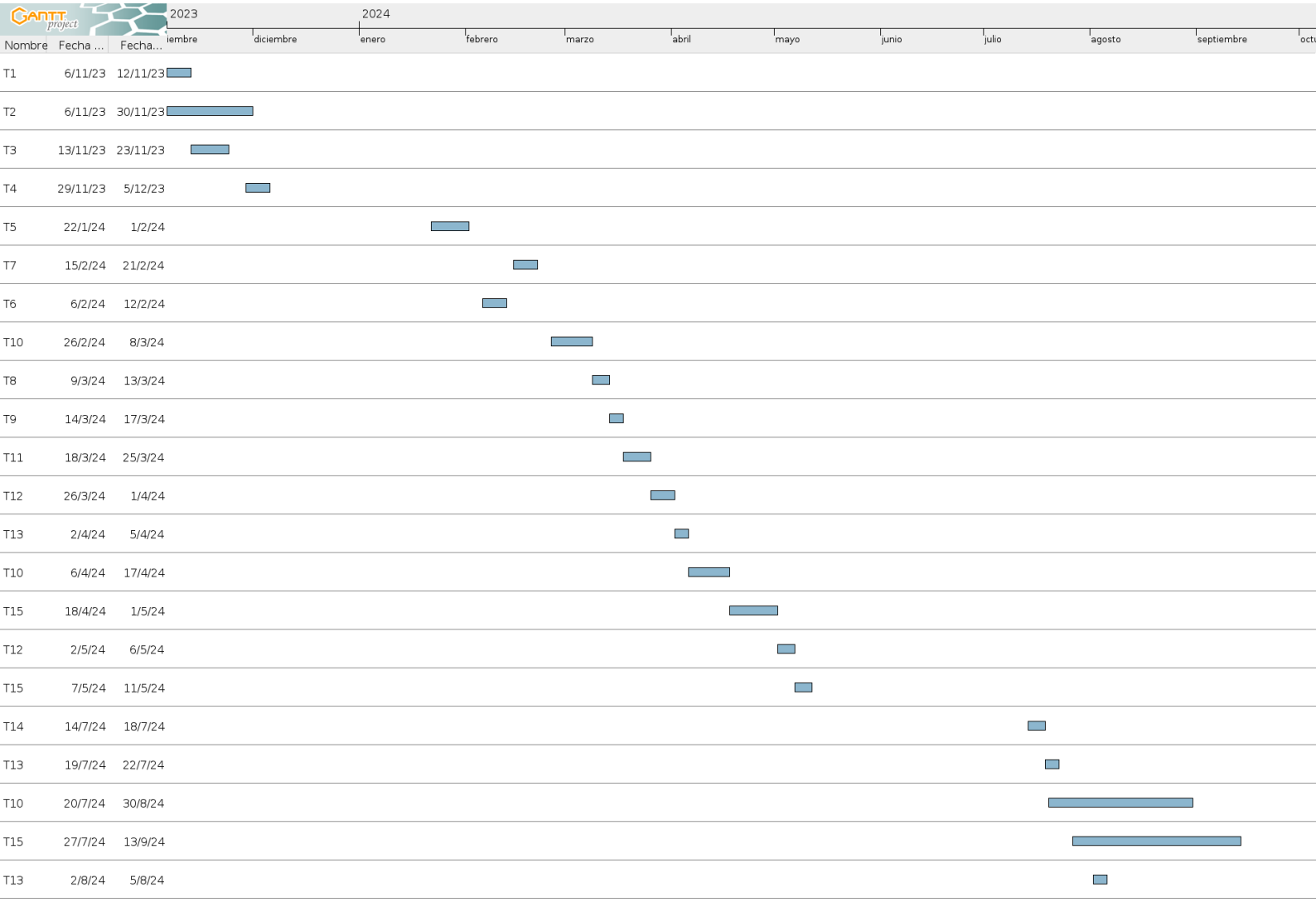

Figura 1.1: Diagrama de Gantt

1.3. Estructura de la memoria

El resto de esta memoria se estructura como sigue:

- El capítulo 2 describe las distintas herramientas utilizadas para realizar este trabajo, con el objetivo de que el lector se familiarice con ellas de cara a capítulos sucesivos.
- El capítulo 3 introduce el soporte *hardware* para particionado que vamos a utilizar, Intel CAT, junto con el desarrollo de la API de particionado de L2 en PMCTrack y de los plugins de particionado de la caché, uno para la plataforma Alder Lake (particionado de caché L2), y otro para la plataforma Broadwell-EP (particionado de caché L3).
- El capítulo 4 incluye los resultados de nuestra experimentación en las distintas plataformas utilizadas, así como un extenso análisis de contadores *hardware* realizado en Broadwell-EP.
- El capítulo 5 resume las conclusiones principales de este trabajo, y las posibles líneas de trabajo futuro.
- Finalmente, los apéndices A y B incluyen la traducción al inglés de los capítulos de introducción y conclusiones.

Capítulo 2

Herramientas Utilizadas

En este capítulo se presentan las distintas herramientas utilizadas a lo largo del trabajo para poder monitorizar el rendimiento, implementar las estrategias de particionado evaluadas y lanzar los experimentos.

2.1. PMCTrack

PMCTrack[18] es una herramienta open-source para Linux que permite la monitorización del rendimiento mediante el uso de contadores hardware. Inicialmente fue desarrollada para facilitar la implementación de algoritmos de planificación en el kernel Linux que hagan uso de información extraída de contadores de monitorización del rendimiento (*Performance Monitoring Counters* - PMCs) para realizar optimizaciones en tiempo de ejecución. Posteriormente se añadieron a PMCTrack distintos componentes de espacio de usuario que permiten al usuario final obtener información de monitorización de aplicaciones en tiempo de ejecución.

Como puede verse en la figura 2.1, PMCTrack está formada por múltiples componentes.

PMCTrack incluye un módulo del kernel Linux, que contiene todas sus funcionalidades, y una aplicación de espacio de usuario que permite acceder a ellas de forma más amigable para el usuario. Por tanto, para poder utilizar PMCTrack es necesario cargar el módulo en el kernel Linux. Si el kernel es anterior a Linux v5.9 será necesario también parchear el kernel [19].

El módulo posee un *core* que es independiente de la arquitectura, sobre cuya API se construyen diversos módulos de monitorización. Uno de estos módulos es PMCSched, del que hablaremos en la sección 2.2 y que admite a su vez sus propios *plugins*. Estos *plugins* permiten exponer al planificador del kernel y al espacio de usuario (a través de **/proc/pmc/mm_manager**) cualquier tipo de información de monitorización, aunque esta no esté disponible mediante los contadores hardware de la PMU (*Performance Monitoring Unit*) del procesador. Esto se hace exportando *contadores virtuales*, que como su nombre indica se comportan como los contadores hardware pero no están presentes en la PMU.

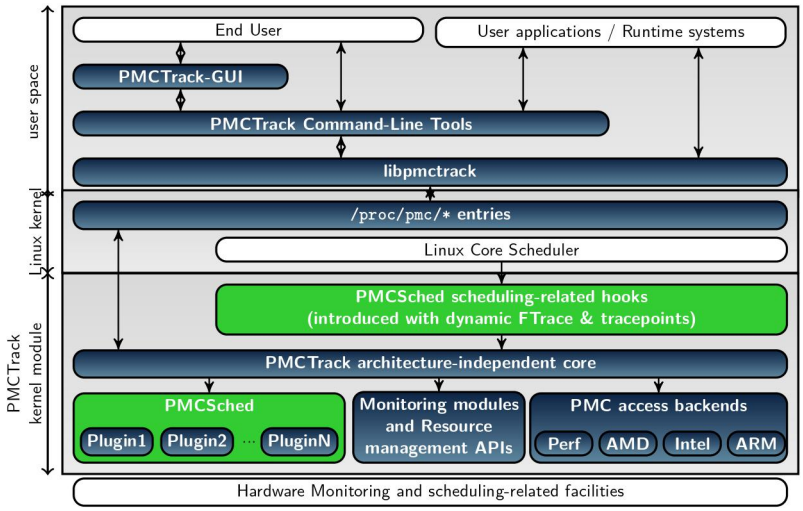

Figura 2.1: Arquitectura de PMCTrack [1]

El módulo del kernel de PMCTrack dispone de un *backend* dependiente de la arquitectura que se encarga de implementar la funcionalidad que ofrece PMCTrack con el procesador en cuestión. Además, instala una serie de *hooks* en el kernel (puntos de traza de `ftrace`), para que el módulo sea consciente de eventos clave de planificación, como los cambios de contexto.

Finalmente, ofrece una serie de interfaces en el directorio **/proc/pmc** que permiten configurar (activar y desactivar plugins, habilitar y deshabilitar el modo verboso, etc) PMCTrack y PMCSched desde espacio de usuario.

En espacio de usuario, PMCTrack ofrece una GUI, una utilidad de línea de comandos (`pmctrack`) y una biblioteca (`libpmctrack`). La GUI es esencialmente un frontend para la utilidad de línea de comandos aunque añade algunas funcionalidades no disponibles en esta, como la posibilidad de monitorear máquinas remotas a través de SSH [20].

La herramienta `pmctrack` de línea de comandos permite recopilar información del módulo del kernel de PMCTrack desde espacio de usuario. Para ilustrar el funcionamiento de la herramienta `pmctrack` consideramos el siguiente ejemplo:

```
$ pmctrack -E -c instr:ebs,llc_misses,llc_references  -V llc_usage
./benchmarks/common/parsec3/blackscholes_t_p3 1 active
[Event-to-counter mappings]
pmc0=instr
pmc3=llc_misses
pmc4=llc_references
virt0=llc_usage
[Event counts]
nsample    pid    event        pmc0     pmc3     pmc4      etime_us         virt0
      1   3401    ebs     500000040    10527    85954       108369        9568256
```

```
  2   3401   ebs   500000083   25   33770   93362   13926400
  3   3401   ebs   500000070    8   33493   93269   16809984
  4   3401   ebs   500000070    7   33762   92980   17661952
  5   3401   ebs   500000063   21   34094   93281   19693568
  6   3401   ebs   500000075   10   33673   93273   20152320
  7   3401   ebs   500000015   45   33467   93285   20447232
  8   3401   ebs   500000077   33   33498   93271   20545536
  9   3401   ebs   500000032    8   33412   93257   20480000
 10   3401   ebs   500000035   59   33661   93287   20578304
 11   3401   ebs   500000050   25   33945   93313   20021248
 12   3401   ebs   500000077   10   33614   93264   20021248
 13   3401   ebs   500000060   42   33738   93280   19922944
 14   3401   ebs   500000076   42   33530   93303   19988480
 15   3401   ebs   500000061    5   33624   93286   19955712
 16   3401   ebs   500000067   18   33883   93294   20119552
 17   3401   ebs   500000051   24   33373   92935   20381696
 18   3401   ebs   500000044   19   33462   93223   20512768
 19   3401   ebs   500000068   16   34003   93234   20512768
 20   3401   ebs   500000048   36   33797   93256   20545536
 21   3401   ebs   500000053    8   33407   93243   20480000
...
```

Este comando proporciona al usuario el número de instrucciones retiradas (desde la última vez que se informó), el número de fallos y referencias de caché de último nivel (LLC), el tiempo transcurrido en μs desde la última muestra (`etime_us`) y el uso de la caché (en bytes). En la salida de la ejecución podemos ver cómo el espacio utilizado de la caché aumenta rápidamente al principio de la ejecución de `blackscholes`. La estructura del comando `pmctrack -E -c instr:ebs,llc_misses,llc_references -V llc_usage ./benchmarks/common/parsec3/blackscholes_t_p3 1 active` es la siguiente: `-E` establece el modo a *event sampling*, es decir, se recopila una muestra cada vez que el valor de un cierto contador alcance un umbral establecido. Alternativamente el modo `-T` seguido de un número X de segundos hace que se obtenga una muestra cada X segundos. `-c` establece la lista de eventos hardware que deben ser proporcionados, indicando con `:ebs=n` aquel que usamos para establecer el umbral n. En nuestro caso es el contador de instrucciones que por defecto tiene como umbral $n = 500\ 000\ 000$ y por tanto se recoge una muestra cada 500 000 000 instrucciones retiradas (aproximadamente). El parámetro `-V` nos permite establecer la lista de eventos virtuales que deben ser monitorizados. Finalmente `./benchmarks/common/parsec3/blackscholes_t_p3 1 active` es el comando a cuya ejecución ha de aplicarse la configuración de contadores anterior, en nuestro caso uno de los *benchmarks* que vamos a introducir en próximos capítulos. La lista de eventos disponibles (para las opciones `-c` y `-V`) puede consultarse mediante el comando `pmc-events -L`. Incluye tantos como los ofrecidos por la PMU (contadores hardware) como los contadores virtuales. Se listan todas las PMUs, si hubiera más de una (por ejemplo, en el caso de un sistema asimétrico, como la plataforma Intel Alder Lake).

Desde espacio de usuario también es posible programar software en C y C++ que aproveche las características de PMCTrack gracias a la biblioteca `libpmctrack`, que simplemente ofrece una interfaz más amigable para las entradas que el driver expone en /**proc**.

2.2. PMCSched

PMCSched es un *framework* de prototipado rápido de algoritmos de planificación para el kernel Linux [1]. Su principal ventaja es que permite implementar estos dentro de un módulo cargable del kernel, en lugar de tener que implementarlos directamente en el kernel. PMCSched es un módulo de monitorización de PMCTrack, lo que permite aprovechar la información de los contadores hardware que proporciona PMCTrack a la hora de implementar políticas de planificación.

PMCSched permite crear *plugins* que implementan las políticas de gestión de recursos (como planificación y particionado de caché). Para crear un *plugin* basta con crear una estructura `sched_ops` e implementar algunas de sus operaciones (no necesariamente todas) en un fichero `.c` independiente. La estructura viene dada por la siguiente definición, que mostramos (por trozos) pues será de utilidad en siguientes capítulos:

```
1 typedef struct sched_ops {
2   sched_policy_mm_t policy;
3   struct list_head link_schedulers;
4   char *description;
5   pmcsched_counter_config_t* counter_config;
6   unsigned long flags; /* To control locking scheme */
```

- `policy`: Una constante que identifique unívocamente da la política de planificación. Se trata de un tipo enumerado.
- `link_schedulers`: Interno de PMCSched, para que el *framework* organice todos los plugins disponibles en una lista enlazada.
- `description`: Una descripción para humanos de la política de planificación (se mostrará si lee el fichero /**proc**/**pmc**/**sched**).
- `counter_config`: Configuración de los contadores hardware (y virtuales) que el plugin desee ofrecer. Puede ser `NULL` si no utiliza contadores hardware y no expone contadores virtuales.

```
7   int (*probe_plugin) (void);
8   int (*init_plugin) (void);
9
10  /* The plugin might free some structures */
11  void (*destroy_plugin) (void);
```

- `probe_plugin`: Devuelve `true` (que en C es cualquier entero distinto de 0) si el *plugin* puede ser cargado (todo el hardware que el plugin necesita está soportado por la máquina).

- `init_plugin`: Inicializa el *plugin* en el momento de su carga.
- `destroy_plugin`: Libera los recursos asociados al *plugin* durante su extracción del kernel.

Las siguientes *callbacks* se llaman por cada hilo activo del sistema, cuando el hilo cumpla cierta condición. Por ejemplo, `on_exec_thread` se llama cuando el hilo ejecuta la llamada al sistema `exec`. El tipo `pmcsched_thread_data_t` es una estructura que guarda la información de cada hilo (aplicación a la que pertenece, estado, etc).

```
12  /* Callbacks to capture basic scheduling events */
13  void (*on_exec_thread) (pmon_prof_t* prof);
14  void (*on_active_thread) (pmcsched_thread_data_t* t);
15  void (*on_inactive_thread)(pmcsched_thread_data_t* t);
16  int (*on_fork_thread) (pmcsched_thread_data_t* t, unsigned char
        is_new_app);
17  void (*on_free_thread) (pmcsched_thread_data_t* t, unsigned char
        is_last_thread);
18  void (*on_exit_thread) (pmcsched_thread_data_t* t);
```

- `on_exec_thread`: Se llama cuando el hilo ejecuta *exec* y recibe como parámetro la información de *profiling* (estructura `pmon_prof_t`) del hilo.
- `on_active_thread`: Se llama cuando el hilo se vuelve activo (es decir, cuando el hilo se marca como listo para ejecutar) y recibe como parámetro la información del hilo.
- `on_inactive_thread`: Como la anterior, pero se llama cuando el hilo sale del estado listo para ejecutar o de ejecución (el hilo se bloquea por E/S, sincronización o termina).
- `on_fork_thread`: Se llama cuando un hilo invoca `fork()`. A diferencia de las anteriores recibe un parámetro `is_new_app` que indica si el hilo pertenece a un nuevo proceso (puede ser falso, por ejemplo, en caso de llamada a `pthread_create`). Devuelve 0 en caso de éxito y distinto de 0 en caso de error.
- `on_free_thread`: Se llama cuando el kernel libera el `task_struct` asociado a un hilo. El `task_struct` es la estructura del kernel Linux para representar hilos.
- `on_exit_thread`: Se llama cuando el hilo termina (tras una llamada a la función estándar `exit()` o por otro motivo, como una división por cero o cualquier otro error que provoque una terminación abrupta).

```
19  /**
20   * Callbacks for per-core-group periodic scheduling
21   * from process context and interrupt context respectively
22   **/
23  void  (*sched_kthread_periodic)
24  (sized_list_t* migration_list);
25  void  (*sched_timer_periodic)(void);
```

- `sched_kthread_periodic`: Se llama periódicamente desde contexto de proceso (desde un kthread). Solo la invocan los *plugins* que realizan migraciones de hilos, la cual es una tarea bloqueante en el kernel. Recibe un puntero a la lista de hilos que van a migrarse.

- `sched_timer_periodic`: También se llama periódicamente, pero desde contexto de interrupción cada vez que expira un temporizador. El periodo son `sched_period_normal` milisegundos, donde `sched_period_normal` es un parametro global de PMCSched configurable a través del fichero **/proc/pmc/sched**.

Ambas se llaman sólo una vez por grupo de núcleos (*core group*), ya que cada grupo de núcleos tiene su propia PMU. Un grupo de núcleos no es más que un conjunto de núcleos que comparten un último nivel de caché. Los *cores* del sistema se asignan a un conjunto dependiendo de su tipo (en sistemas asimétricos) o de su disposición en la topología del sistema. En nuestra plataforma experimental Alder Lake (que es un sistema asimétrico) disponemos de dos grupos de procesadores, uno que contiene los procesadores rápidos y otro que contiene los eficientes.

```
26  /**
27   * Callback invoked when performance counters are sampled.
28   */
29  int (*on_new_sample) (pmon_prof_t* prof, int cpu,
30                        pmc_sample_t* sample,int flags,void* data);
31
32  /* Read and write plugin's configurable parameters */
33  int (*on_read_plugin) (char *aux);
34  int (*on_write_plugin) (char *line);
```

- `on_new_sample`: Se llama cada vez que se genera una muestra de los contadores hardware. Recibe entre otros argumentos la información de *profiling* del hilo (que contiene un puntero a la estructura `pmcsched_thread_data_t` del hilo) y un puntero `sample` a la estructura que contiene los datos de la muestra y en la que han de rellenarse los contadores virtuales (en caso de que el plugin exporte alguno). Como es habitual devuelve 0 en caso de éxito y distinto de 0 en caso de error.

Si el plugin está activado en **/proc/pmc/sched**, las llamadas de lectura y escritura a este pseudo-fichero provocan a su vez una llamada a las siguientes funciones:

- `on_read_plugin`: Escribe en `aux` una cadena de texto que se mostrará al final de **/proc/pmc/sched**. Suele tratarse de información sobre el estado de configuración del plugin (por ejemplo, en el caso de un plugin de particionado de caché podría incluir las particiones y las vías asociadas a cada partición).

- `on_write_plugin`: Recibe en el parámetro `line` una cadena de texto. Habitualmente se utiliza para pasar al plugin información de configuración.

```
35  /**
36   * Callbacks for controlling
37   * low-level scheduling events in plugins
38   **/
39  void (*on_switch_in_thread)(pmon_prof_t* prof,
40                              pmcsched_thread_data_t* t, unsigned char
                                    prof_enabled);
41
42  void (*on_switch_out_thread)(pmon_prof_t* prof,
43                               pmcsched_thread_data_t* t, unsigned char
                                     prof_enabled);
44
45  void (*on_migrate_thread)(pmcsched_thread_data_t* t, int prev_cpu,
        int new_cpu);
46
47  void (*on_tick_thread)(pmcsched_thread_data_t* t, int cpu);
```

- `on_switch_in_thread`: Se llama justo antes de que el hilo entre a ejecutar en una CPU. Recibe la información de *profiling*, la estructura `pmcsched_thread_data_t` del hilo y un booleano que indica si el *profiling* está habilitado para el hilo (los hilos que no han sido lanzados por el usuario tienen el *profiling* desactivado).
- `on_switch_out_thread`: Idéntica a la anterior, pero se llama cuando el hilo se acaba de ser expulsado de una CPU.
- `on_migrate_thread`: Similar, cuando un hilo se mueve de una CPU a otra.
- `on_tick_thread`: Se llama en cada "tick", es decir, cada vez que el planificador actualiza estadísticas tras recibir una interrupción de reloj.

2.3. Het-Harness

Het-Harness es una herramienta que simplifica el proceso de lanzar experimentos con múltiples cargas en sistemas multiprocesador. *Het-Harness* está formada por en una serie de scripts, programas en C y C++, librerías y archivos de configuración. Este framework se diseñó inicialmente para ayudar en la evaluación experimental de políticas de planificación para sistemas asimétricos (como el Alder Lake), aunque ahora también soporta sistemas simétricos. Esta herramienta es propietaria.

El framework funciona sobre dos abstracciones: *benchmarks* y *benchsets*. Un *benchmark* es una aplicación que puede ser lanzada como parte de una carga de trabajo o *workload* (un conjunto de una o más aplicaciones que van a ser lanzadas simultáneamente). Un *benchset* es una sucesión de *workloads* que serán lanzadas una tras otra.

Actualmente Het-Harness dispone de un amplio abanico de scripts para benchmarks de las suites más populares incluyendo SPEC CPU2000, CPU 2006, CPU 2017, SPEC OMP01, OMP12, PARSEC 2 y 3, NAS Parallel Benchmarks (NPB),

Rodinia, etc. No obstante, Het-Harness sólo incluye los scripts para ejecutar estos benchmarks, no los ejecutables de los benchmarks en sí (ni sus datos de entrada). Estos deberán haber sido adecuadamente situados por el usuario en `$HOME/Benchmarks`.

Durante la ejecución de un benchset, Het-Harness va generando un fichero de log por cada workload individual del benchset que termina de ejecutarse. El archivo se almacena en `/var/tmp/${USER}/het-results` y contiene tanto el tiempo que tardó el programa en ejecutarse como otra información general (número de cores activos en el sistema, frecuencia de cada core, etc).

Además, Het-Harness incluye scripts que permiten parsear fácilmente estos archivos de log, para generar resúmenes, calcular medias, métricas, etc.

Capítulo 3

Introducción a Intel Resource Director Technology

En este capítulo introducimos las plataformas sobre las que vamos a realizar el estudio. A continuación presentamos la tecnología Intel RDT, que nos da el soporte hardware necesario para implementar políticas de particionado de caché. Finalmente mostramos la implementación de la API de PMCTrack para el particionado de la caché $L2$ utilizando esta tecnología y la implementación de los *plugins* de PMCSched (uno para cada plataforma) que utilizaremos para el particionado de caché durante los experimentos.

3.1. Plataformas utilizadas

3.1.1. Intel Alder Lake

Alder Lake es el sobrenombre de la duodécima generación de procesadores de Intel. Presentan una arquitectura multicore asimétrica que combina núcleos rápidos (llamados P-cores, de *performance cores*) pero poco eficientes (en términos de la energía consumida) con procesadores más lentos y eficientes (llamados E-cores, de *efficient cores*). Esta es el primer procesador asimétrico comercial de Intel para escritorio, similar a *ARM big.LITTLE* [21]. El objetivo de estas arquitecturas es disponer de procesadores multi-core capaces de adaptarse dinámicamente a las necesidad de cómputo siendo más eficientes que un simple escalado dinámico de frecuencia. De esta forma pueden optimizarse simultáneamente consumo y rendimiento en un mismo sistema [22].

A lo largo de este trabajo utilizamos concretamente el Intel Core i9-12900K (microarquitectura Alder Lake), que dispone de 8 P-cores y 8 E-cores y cuya topología se presenta gráficamente en la figura 3.1. Como podemos ver, los E-cores (cores $8-15$) están agrupados en clústers de 4 cores que comparten una caché $L2$ de $2MB$. Por contra cada P-core (cores $0-7$) dispone de una caché $L2$ privada de $1{,}25MB$. Además, cada procesador dispone de una caché privada $L1$ y todos los procesadores comparten una caché $L3$ de $30MB$ (que además es la caché de último nivel, LLC). El sistema operativo utilizado fue Debian GNU/Linux 11, con el kernel

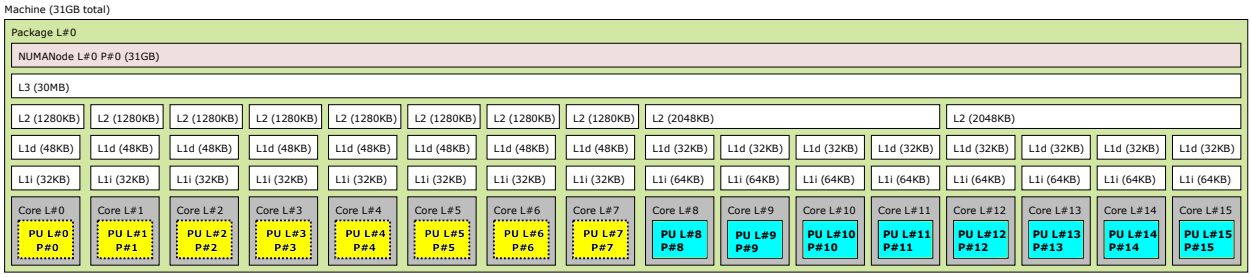

Figura 3.1: Topología de la plataforma Alder Lake [2].

Linux v5.16.9. Utilizamos la versión "*vanilla*" del kernel, es decir, la versión sin modificar que se publica en `http://kernel.org`. Este será el sistema que utilizaremos inicialmente para nuestros experimentos.

3.1.2. Broadwell-EP

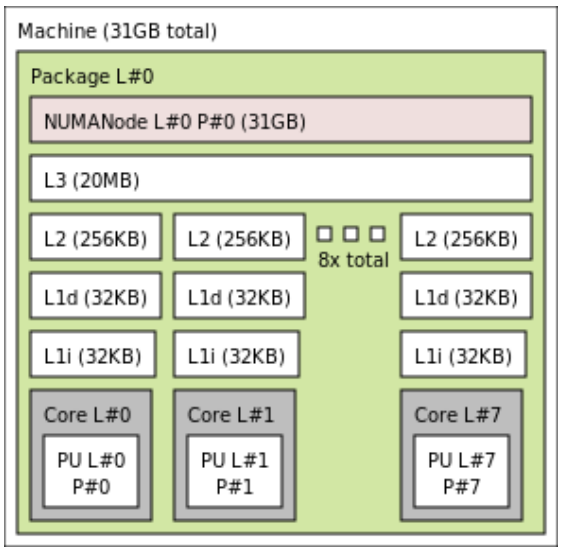

Figura 3.2: Topología de la plataforma Broadwell-EP [3].

Lanzada en 2016, la microarquitectura Broadwell-EP pertenece a la quinta generación de procesadores de Intel (Broadwell) [23]. En este caso tenemos una arquitectura homogénea, y en nuestra máquina disponemos de un procesador Intel Xeon CPU E5-2620 v4 con 8 núcleos con una frecuencia máxima de 2,1 GHz, cuya topología se muestra en la figura 3.2. Cada núcleo dispone de una caché privada de datos y otra de instrucciones ($L1$, de 256 KiB cada una) y una caché privada $L2$ de 2 MiB. La caché $L3$ de último nivel tiene una capacidad de 20 MiB y 20 vías y es compartida por todos los procesadores.

El sistema operativo utilizado en esta máquina fue Debian GNU/Linux 12, también con la versión *vanilla* del kernel Linux v6.2.4 (más moderno que el usado en

la plataforma Alder Lake, con el objetivo de que los resultados obtenidos sean más relevantes).

3.2. Intel Resource Director Technology (RDT)

Intel RDT proporciona soporte *hardware* para monitorización y asignación de recursos de memoria compartida en algunos procesadores multinúcleo de Intel. Este trabajo explora el soporte para particionado de caché de *Intel RDT*.

Intel RDT ofrece una serie de capacidades para asignar los recursos de la caché, fundamentalmente *Cache Allocation Technology (CAT)* y *Code and Data Prioritization (CDP)*. El primero permite el particionado de caché, mientras el segundo diferenciar entre código y datos a la hora de realizar el particionado. La tecnología *Intel CAT* está disponible tanto en la plataforma Alder Lake que hemos utilizado como en Broadwell-EP, y su evaluación es el principal objetivo de este trabajo. La información se encuentra en la sección 18.19 del Volumen 3 del manual de Intel [4].

3.2.1. Intel Cache Allocation Technology (CAT)

Intel CAT permite a un sistema operativo o hipervisor especificar la cantidad de memoria caché que una aplicación puede utilizar. Dependiendo de la familia del procesador el particionado está disponible para la caché $L2$ o la caché $L3$. En el caso de Alder Lake lo que podemos particionar es la caché $L2$, mientras que la plataforma Broadwell-EP solo soporta el particionado de $L3$.

El *software* puede determinar si el hardware soporta *Intel CAT* y qué características concretas tiene utilizando `cpuid`, como veremos en la sección siguiente. El particionado de caché puede modificarse dinámicamente en cualquier momento de la ejecución escribiendo a registros MSR (*Model Specific Register*, registros de control disponibles en la arquitectura `x86` que permiten al sistema operativo configurar aspectos del *hardware* mediante lecturas y escrituras de dichos registros [24]).

El procesador expone un conjunto de *classes of service* (COS). Cada una se corresponde con una partición de la caché, y viene dada por una máscara de bits llamada *capacity bitmask* (CBM). Cada *core* de la máquina tiene un COS asignado en cada momento que determina qué partición aplica a los hilos que se ejecutan en dicho *core*, como ilustra la figura 3.3. Cabe destacar que el particionado solo se tiene en cuenta a la hora de reemplazar líneas. Los hilos pueden acceder a toda la caché para realizar lecturas y escrituras [4].

Las máscaras (CBM) de COS distintas no tienen porqué ser disjuntas, es decir, se permite que haya solapamientos entre clases. En cuanto a la forma de asignar el COS: **cada core** expone un registro, `IA32_PQR_ASSOC` que permite al sistema operativo o hipervisor especificar el COS del hilo planificado para ejecutarse en dicho core. Las máscaras pueden configurarse escribiendo al registro MSR `IA32_resourceType_MASK_n` donde `resourceType` es `L2` o `L3` y `n` indica el número de clase de servicio (COS). La longitud de las máscaras depende de la máquina (es parte de la información que se obtiene con `cpuid`). Además, deben ser todas

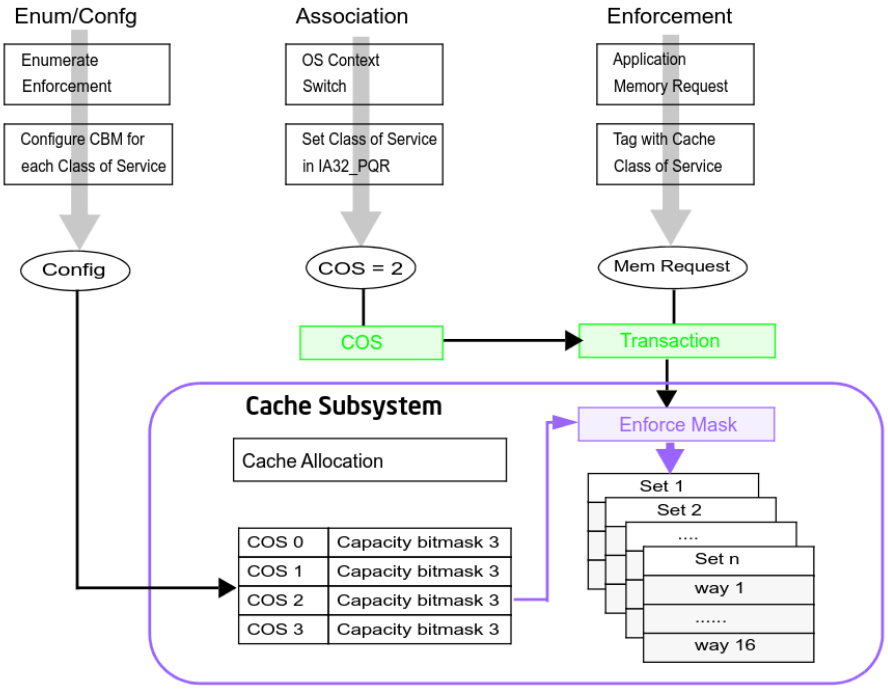

Figura 3.3: Funcionamiento de COS y CBM, ilustrado. Figura procedente de [4]

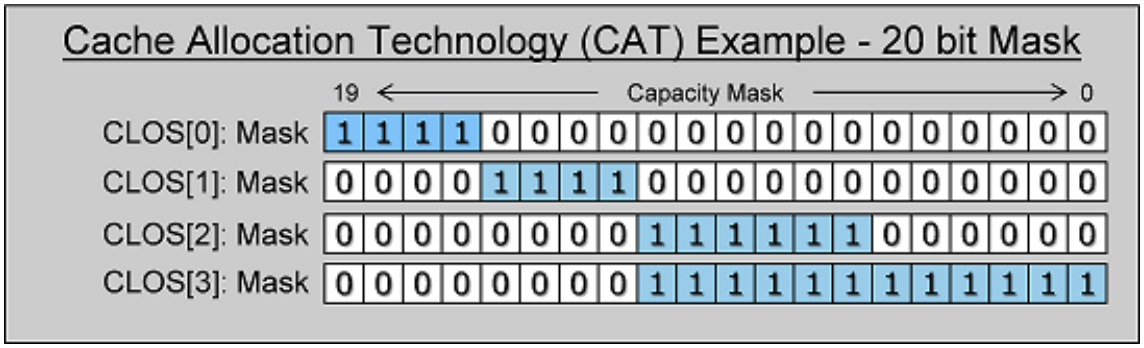

Figura 3.4: Ejemplo de máscaras válidas para particionado de caché con Intel CAT.

combinaciones contiguas de '1', salvo que `cpuid` garantice que existe soporte para particionado no contiguo. La figura 3.4 contiene ejemplos de máscaras contiguas. El bit i-ésimo de la máscara indica si la vía i-ésima forma parte de la partición o no.

3.3. API de Intel Cache Allocation Technology (L2) para PMCTrack

Para poder desarrollar plugins de particionado para la caché $L2$ en PMCSched necesitamos añadir a PMCTrack el soporte de particionado de la $L2$ (el de la $L3$ ya está implementado). A continuación presentamos la API desarrollada en este trabajo, que se encuentra declarada en el fichero **include/pmc/intel_rdt.h**:

```
1 /* CAT SPECIFIC stuff */
```

```
typedef struct {
  unsigned int cat_nr_cos_available;
  unsigned int cat_cbm_length;
  unsigned int cat_cbm_mask;
  unsigned int cat_shareable_mask;
  unsigned char core_type; /* For Hybrid Processors */
} intel_l2cat_support_t;

int l2_cat_probe(void);
int l2_cat_initialize(intel_l2cat_support_t* l2_cat_support, int
    nr_core_types);
int l2_cat_release(intel_l2cat_support_t* l2_cat_support);

int intel_l2_cat_set_capacity_bitmask(intel_l2cat_support_t*
    cat_support, unsigned int cosid, unsigned int mask);

unsigned int intel_l2_cat_get_capacity_bitmask(intel_l2cat_support_t*
    cat_support, unsigned int cosid);
```

La estructura `intel_l2cat_support_t` almacena la información sobre las capacidades del particionado de caché, que son devueltos por `cpuid`:

- `cat_nr_cos_available`: Número de clases de servicio disponibles, es decir, número máximo de particiones distintas simultáneas.
- `cat_cbm_length`: Longitud de la máscara que define una clase de servicio.
- `cat_cbm_mask`: Máscara de bits para una partición que ocupa toda la caché (máscara por defecto).
- `cat_shareable_mask`: Contiene una máscara en los `cat_cbm_length` bits menos significativos que indica las vías que están compartidas con otros recursos hardware (dispositivos entrada/salida como PCIe, etc). Estas vías no pueden ser seleccionadas para formar una partición si no hay al menos una vía con el bit correspondiente a 0 [25].
- `core_type`: Indica a qué tipo de core corresponde la información del `struct`. En la plataforma Alder Lake, necesitamos dos estructuras `intel_l2cat_support_t`: Una para los P-cores y otra para los E-cores, puesto que la caché L2 de los P-cores es más pequeña (tiene menos vías) que las de los E-cores y por tanto van a ser necesarios valores distintos de `cat_cbm_mask` y `cat_cbm_length` en cada tipo de *core*.

La implementación de las funciones anteriores viene dada por:

```
int l2_cat_probe(void){
  cpuid_regs_t cpuid_regs;

  if (boot_cpu_data.x86_vendor!=X86_VENDOR_INTEL)
    return -ENOTSUPP;

  /**
   * Instructions for Intel L2 CAT Detection
```

```
    * can be found in Section 18.19.4.2 of IntelSDM
    * Cache Allocation Technology: Resource Type and Capability
       Enumeration
    **/

   /* Check capabilities*/
   cpuid_regs.eax=0x10;
   cpuid_regs.ecx=0x0;
   cpuid_regs.ebx=cpuid_regs.edx=0x0;

   run_cpuid(cpuid_regs);

   if (!(cpuid_regs.ebx & (1<<2)))
     return -ENOTSUPP;

   return 0;

}
```

Esta rutina comprueba si la máquina dispone del hardware necesario para particionado L2 (Intel CAT). Para ello verifica simplemente que el fabricante de la CPU sea Intel (condición necesaria para que pueda soportar el particionado de Intel) y en tal caso llama a la función *leaf* 10, subfunción *leaf* 0. Esto se hace asignando los registro `EAX=0x10` y `ECX=0x0`, y entonces se llama a la instrucción máquina `cpuid`, que devuelve en el registro EBX información sobre las capacidades de particionado de la máquina. En la figura 3.5 se muestra la interpretración que se debe dar a cada bit. El segundo bit menos significativo indica si existe soporte de particionado $L3$ o no, y el tercer bit menos significativo si existe soporte de particionado $L2$ (lo que en el código comprobamos con `!(cpuid_regs.ebx & (1<<2))`). Se devuelve 0 si el soporte de caché $L2$ está presente y error (no soportado, `ENOTSUPP`) en otro caso.

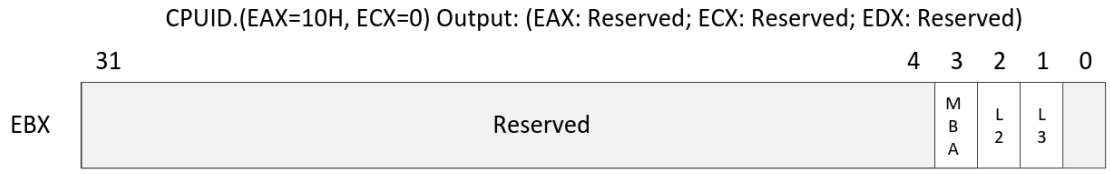

Figura 3.5: EBX devuelto por `cpuid`, llamado con `EAX=0x10, ECX=0`. Figura procedente de [4]

El manual también nos indica que el número de bit (en el caso de la $L2$, el 2, como se muestra en la figura 3.5) se corresponde el número de la subfunción de `cpuid` que devuelve la información específica de cada soporte de caché. Es decir, que para rellenar la estructura `intel_l2_cat_support_t` tenemos que hacer lo siguiente (seleccionamos la subfunción de `cpuid` mediante el registro ECX):

```
static void l2_cat_populate_capabilities(void *data)
{
   intel_l2cat_support_t* l2_cat_support=data;
   cpuid_regs_t cpuid_regs;

```

```
6    /**
7     * Instructions for Intel L2 CAT Detection
8     * can be found in Section 18.19.4.2 of IntelSDM
9     * Cache Allocation Technology: Resource Type and Capability
          Enumeration
10    **/
11
12   /* Check capabilities*/
13   cpuid_regs.eax=0x10;
14   cpuid_regs.ecx=0x2;
15   cpuid_regs.ebx=cpuid_regs.edx=0x0;
16
17   run_cpuid(cpuid_regs);
18
19   l2_cat_support->cat_cbm_length=(cpuid_regs.eax & 0x1f)+1;
20   l2_cat_support->cat_shareable_mask=cpuid_regs.ebx;
21   l2_cat_support->cat_cbm_mask=(1<<l2_cat_support->cat_cbm_length)-1;
22   l2_cat_support->cat_nr_cos_available=(cpuid_regs.edx & 0xffff)+1;
23 }
```

La figura 3.6 muestra los valores de retorno tras la llamada a `cpuid`. La longitud de la máscara (es decir, el número de vías disponibles) se obtiene quedándonos con los 5 bits menos significativos del registro EAX y sumando 1 (de esta forma se puede aprovechar el valor 0 para denotar una longitud de máscara válida). La máscaras de vías compartidas se obtiene del valor del registro EBX. El número de clases de servicio (COS) disponibles se obtienen sumando 1 al valor de los 16 bits menos significativos del registro EDX.

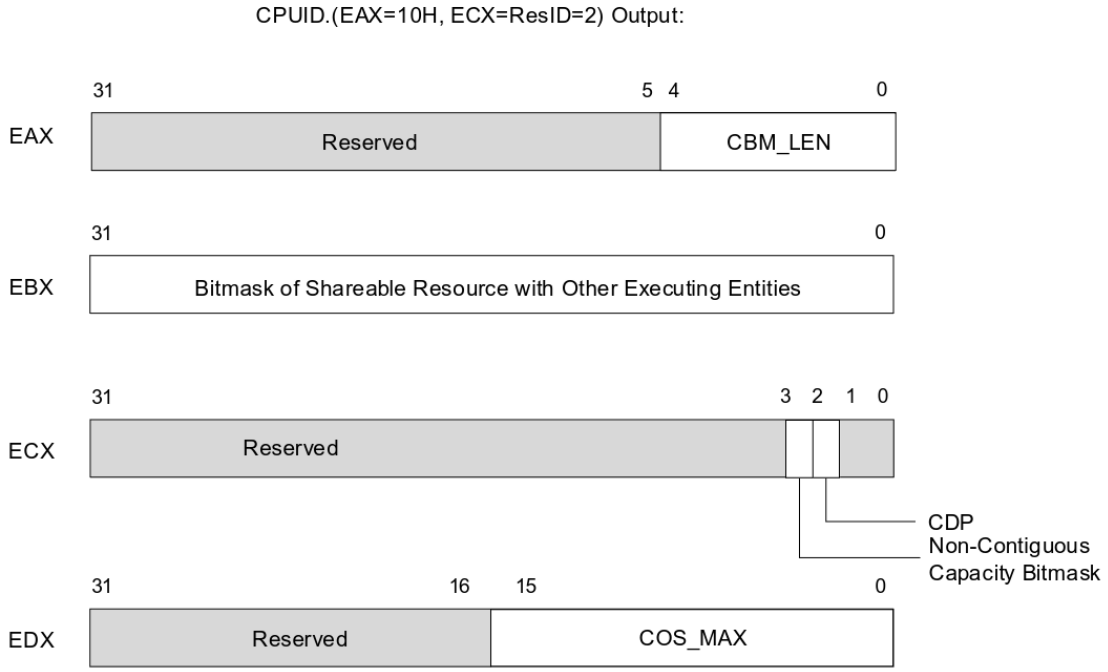

Figura 3.6: Valores de retorno de `cpuid` (`EAX=0x10, ECX=0x2`). Figura procedente de [4]

Como hemos dicho antes, en un sistema asimétrico (como Alder Lake) tenemos que tener en cuenta que *cores* diferentes pueden tener soportes de particionado diferentes:

```
/* Make this asymmetry-aware */
int l2_cat_initialize(intel_l2cat_support_t* l2_cat_support, int
    nr_core_types){
  int i;

  if (get_nr_coretypes()!=nr_core_types)
    return -ENOTSUPP;

  /* Initialize pointer */
  private_l2cat_support=l2_cat_support;

  for (i=0;i<nr_core_types;i++){
    int cpu=get_any_cpu_coretype(i);

    /* Issue query to any of the cores of the corresponding type */
    smp_call_function_single(cpu, l2_cat_populate_capabilities, (void*)
       &private_l2cat_support[i], 1);

    private_l2cat_support[i].core_type=i;
  }

  return 0;
}
```

Esta función simplemente inicializa una estructura `intel_l2_cat_support_t` por cada tipo de *core* distinto, ejecutando `l2_cat_populate_capabilities` en una CPU de cada tipo. Si el número de tipos de núcleos que el *plugin* de PMCSched espera es distinto que el presente en la máquina se aborta.

```
int l2_cat_release(intel_l2cat_support_t* l2_cat_support){
  private_l2cat_support=NULL;
  return 0;
}
```

`l2_cat_release` borra el contenido del puntero `private_l2cat_support`. Este puntero es una variable global que sirve para que sólo un *plugin* pueda usar el particionado de caché en un momento dado.

Finalmente tenemos las rutinas que se encargan de leer y escribir máscaras de particionado:

```
int intel_l2_cat_set_capacity_bitmask(intel_l2cat_support_t*
    cat_support, unsigned int cosid, unsigned int mask){

  if (cosid<0 || cosid>=cat_support->cat_nr_cos_available)
    return -EINVAL;

  mask&=cat_support->cat_cbm_mask;
  wrmsr(IA32_L2_MASK_0+cosid, mask, 0);
  return 0;
}

```

```
11 unsigned int intel_l2_cat_get_capacity_bitmask(intel_l2cat_support_t*
       cat_support, unsigned int cosid){
12   uint64_t val;
13
14   if (cosid<0 || cosid>=cat_support->cat_nr_cos_available)
15     return -EINVAL;
16
17   rdmsrl(IA32_L2_MASK_0+cosid, val);
18   val&=cat_support->cat_cbm_mask;
19
20   return (unsigned int) val;
21 }
```

Estas funciones simplemente utilizan las funciones `rdmsrl` y `wrmsr` para leer y escribir respectivamente de un registro MSR. En nuestro caso partimos del registro "base" `IA32_L2_MASK_0` (`0xd10`) y sumamos `i` para calcular `IA32_L2_MASK_i`.

3.4. Plugins de particionado

3.4.1. Alder Lake

La caché L2 de la plataforma Alder Lake tiene 11 vías, en el caso de las cachés L2 privadas que tiene cada P-core, y 16 en el caso de las cachés L2 que comparten cada 4 E-cores.

La política de particionado que vamos a seguir es sencilla, o bien la caché no se particiona, o bien se reparte completamente a partes iguales entre todos los *cores*. Las particiones se asignan directamente a cada *core*, en vez de a cada aplicación (es decir, cada *core* tiene un COS ID fijo). Esto provoca que la caché quede particionada a nivel de hilo, pues cada hilo de una aplicación se ejecutará en un procesador distinto con una partición asignada distinta de las del resto de hilos, con lo que nos permitirá realizar experimentos de particionado de caché a nivel de hilo. A continuación se muestran y se comentan las partes más relevantes del código del *plugin* de particionado:

```
6  typedef enum {
7      L2CAT_CFG_DEFAULT,
8      L2CAT_CFG_SHARED,
9      L2CAT_CFG_PRIVATE,
10     L2CAT_CFG_USER,
11     NR_L2_CAT_CFGS
12 } l2_cat_config_t;
```

Estas son las constantes que definen los modos de funcionamiento del *plugin*. Como vemos hasta aquí, el plugin admite 4 modos de funcionamiento:

- `default`: Se utiliza el COS 0 para todas las aplicaciones, en todos los cores, pero se usa la máscara `0x3ff` en los E-cores, que es la que reporta el hardware para los P-cores. Esto se debe a que el soporte software disponible en el kernel Linux asume erróneamente que el soporte de caché tiene la misma configuración en todos los *cores* [26] (y por tanto, que todas las máscaras son

idénticas). Esta máscara es inadecuada pues deja varias vías sin utilizar en los E-cores (hay 16 vías por cada *socket* y aquí sólo se asignan 10).

- `shared`: Lo mismo que `default`: Caché compartida (todas las aplicaciones van al COS 0) pero usa toda la caché en los E-cores (máscara `0xffff`).
- `private`: Particiona la caché compartida de los E-cores, asignando particiones equitativas y disjuntas a nivel de core. Cada core tiene un COS asociado (COS 1, COS 2, COS 3 y COS 4) y las aplicaciones que corren en cada core lo hacen bajo ese COS. A su vez cada una de esas clases de servicio tiene asociada una partición (dada, respectivamente, por las máscaras `0xf, 0xf0, 0xf00, 0xf000`).
- `user`: Cuando las aplicaciones se ejecutan en un P-core lo hacen bajo COS 1. En los E-cores se tienen los mismos COS ID que en el caso anterior. Se puede especificar desde espacio de usuario las máscaras a utilizar para cada COS.

Los modos que vamos a utilizar para nuestros experimentos son `shared` y `private`.

```
29 static const char* l2_cat_config_str[NR_L2_CAT_CFGS]={"default","shared
     ","private","user"};
30
31
32 static struct {
33   l2_cat_config_t l2_cat_config;
34 } l2cat_plugin_config;
35
36 static intel_l2cat_support_t l2_cat_support[AMP_CORE_TYPES];
37
38 struct user_l2cat_cfg_cpu {
39   unsigned int cos_id; /* Ignored by default */
40   unsigned int mask;
41 };
42
43 #define MAX_L2_CAT_CPUS 24
44 struct user_l2cat_cfg_cpu user_l2cat_cfg[MAX_L2_CAT_CPUS];
45
46
47 static void update_l2_configuration_cpu(void* dummy){
48   int cpu=smp_processor_id();
49     int this_coretype=get_coretype_cpu(cpu);
50     unsigned int ecore_id=(cpu%4);
51     /* WARNING: THIS IS ALDER LAKE SPECIFIC!!! */
52     unsigned int private_mask_ecores[]={0xf, 0xf0, 0xf00, 0xf000};
53   struct user_l2cat_cfg_cpu* l2cat_user=&user_l2cat_cfg[cpu];
54     intel_l2cat_support_t* cur_support=&l2_cat_support[this_coretype];
55
56     switch (l2cat_plugin_config.l2_cat_config)
57     {
```

La función `update_l2_configuration_cpu` se encarga de actualizar la *capacity bitmask* del COSID asociado a la CPU que la está ejecutando, teniendo en cuenta el modo del *plugin*:

```
58          case L2CAT_CFG_DEFAULT:
59              if ((this_coretype==AMP_FAST_CORE) || (ecore_id==0))
60                  intel_l2_cat_set_capacity_bitmask(&l2_cat_support[
                        AMP_FAST_CORE],0,l2_cat_support[AMP_FAST_CORE].
                        cat_cbm_mask);
61              break;
```

Como hemos visto en la sección anterior, `intel_l2_cat_set_capacity_bitmask` se encarga de poner la máscara (`0x3ff` en nuestro caso) en el COS ID (0 en nuestro caso) que se le pase como parámetro. Esa línea de código sólo es ejecutada por los procesadores "*fast*" (los P-cores) y por el procesador 0 de cada *socket* de E-cores. De esta forma se ejecuta una vez por cada caché L2 presente en la topología.

```
62          case L2CAT_CFG_SHARED:
63              if ((this_coretype==AMP_FAST_CORE) || (ecore_id==0))
64                  intel_l2_cat_set_capacity_bitmask(cur_support,0,
                        cur_support->cat_cbm_mask);
65              break;
```

En el modo `shared` el COS ID es de nuevo 0, y la máscara `0x3ff` en los P-cores y `0xffff` en los E-cores (basta asignar `cur_support->cat_cbm_mask`).

```
66          case L2CAT_CFG_PRIVATE:
67              if (this_coretype==AMP_FAST_CORE){
68                  intel_l2_cat_set_capacity_bitmask(cur_support,0,
                        cur_support->cat_cbm_mask);
69              } else {
70                  intel_l2_cat_set_capacity_bitmask(cur_support,ecore_id
                        +1,private_mask_ecores[ecore_id]);
71                  if (ecore_id==0){
72                      intel_l2_cat_set_capacity_bitmask(cur_support,0,
                            cur_support->cat_cbm_mask);
73                  }
74              }
75              break;
```

En el modo `private` los P-cores tienen asignado COS ID 0, mientras que los E-cores tienen COS ID `ecore_id+1`. En este caso asignamos como máscara `private_mask_ecores[ecore_id]`, que consiste en las máscaras `0xf`, `0xf0`, `0xf00`, y `0xf000` las cuales dividen la caché equitativamente (y de forma disjunta) entre los 4 *cores* de cada grupo.

Finalmente, un *core* de cada grupo asigna al COS ID 0 toda la caché (Es el COS ID en el que corren las aplicaciones a las que no afecta el particionado, como aplicaciones del sistema, o en general aplicaciones que no se ejecuten mediante HetHarness, `pmc-launch` o la utilidad de espacio de usuario de `pmctrack`).

```
76          case L2CAT_CFG_USER:
77              if (this_coretype==AMP_FAST_CORE){
78                  intel_l2_cat_set_capacity_bitmask(cur_support,1,
                        l2cat_user->mask);
79              } else {
80                  intel_l2_cat_set_capacity_bitmask(cur_support,ecore_id
                        +1,l2cat_user->mask);
81                  if (ecore_id==0){
```

```
82                      intel_l2_cat_set_capacity_bitmask(cur_support,0,
                            l2_cat_support[AMP_FAST_CORE].cat_cbm_mask);
83                  }
84              }
85              break;
```

Por último, en el modo `user` los P-cores establecen `l2_cat_user->mask`, la cual ha sido proporcionada por el usuario al establecer el modo, como máscara de particionado del COS 1. En el caso de los E-cores se establece, en cada uno, como máscara de particionado del COS `ecore_id`. Además, el COS 0 cubre toda la caché. Cabe notar que necesitamos establecer la máscara del COS 0 únicamente en el caso de los E-cores puesto que esta máscara toma un valor diferente (`0x3ff`) en el modo `default`.

```
87          default:
88              break;
89      }
90  }
91
92
93  static void update_l2_configuration(void){
94      on_each_cpu(update_l2_configuration_cpu, NULL, 1);
95  }
```

`update_l2_configuration` se llama al inicializar y destruir el *plugin* y cada vez que se cambia de modo. Simplemente ejecuta `update_l2_configuration_cpu` en todas las CPUs.

```
100
101 static unsigned int get_cos_id(unsigned int cpu){
102     int this_coretype=get_coretype_cpu(cpu);
103
104     switch (l2cat_plugin_config.l2_cat_config)
105     {
106         case L2CAT_CFG_DEFAULT:
107         case L2CAT_CFG_SHARED:
108             return 0;
109             break;
110         case L2CAT_CFG_PRIVATE:
111             if (this_coretype==AMP_FAST_CORE)
112                 return 0;
113             else
114                 return (cpu%4)+1;
115             break;
116         case L2CAT_CFG_USER:
117             if (this_coretype==AMP_FAST_CORE)
118                 return 1;
119             else
120                 return (cpu%4)+1;
121             break;
122         default:
123             return 0;
124             break;
125     }
```

```
126     return 0;
127 }
```

`get_cos_id` devuelve el COS asociado a una CPU. En los modos `default` y `shared` es 0, mientras que en `private` es 0 para los P-cores, mientras para los E-cores viene dado por `(cpu%4)+1` (que hace corresponder a `cpu` el COS 1, 2, 3 o 4, en consonancia con lo que hemos comentado al describir el modo `private`).

```
129 static int init_plugin_l2cat(void)
130 {
131     char buf[256];
132     char* dest=buf;
133     int i;
134     intel_l2cat_support_t* cur;
135 
136   l2cat_plugin_config.l2_cat_config=L2CAT_CFG_DEFAULT;
137 
138   dest+=sprintf(dest,"Loading L2CAT Manager\n");
139 
140   l2_cat_initialize(l2_cat_support,AMP_CORE_TYPES);
141 
142   for (i=0;i<AMP_CORE_TYPES;i++){
143     cur=&l2_cat_support[i];
144     dest+=sprintf(dest,"[%d]nr_cos=%d\n",i,cur->cat_nr_cos_available);
145     dest+=sprintf(dest,"[%d]mask=0x%x\n",i,cur->cat_cbm_mask);
146   }
147 
148     trace_printk("%s",buf);
149 
150     /* Initialize user settings */
151     for (i=0;i<num_active_cpus();i++){
152         struct user_l2cat_cfg_cpu* user_cfg=&user_l2cat_cfg[i];
153         int this_coretype=get_coretype_cpu(i);
154         intel_l2cat_support_t* cur_support=&l2_cat_support[
                this_coretype];
155 
156         if (this_coretype==AMP_FAST_CORE)
157             user_cfg->cos_id=1;
158         else
159             user_cfg->cos_id=(i%4)+1;
160         user_cfg->mask=cur_support->cat_cbm_mask;
161     }
162 
163     update_l2_configuration();
164 
165   return 0;
166 }
167 
168 static void destroy_plugin_l2cat (void)
169 {
170     /* Restore defaults */
171     l2cat_plugin_config.l2_cat_config=L2CAT_CFG_DEFAULT;
172     update_l2_configuration();
173 
174     l2_cat_release(l2_cat_support);
175 }
```

Las funciones `init_plugin_l2cat` y `destroy_plugin_l2cat` se encargan de inicializar y desinicializar el *plugin* durante su carga y descarga, respectivamente.

```
177 static void on_switch_in_l2cat(pmon_prof_t* prof,
        pmcsched_thread_data_t* t, unsigned char prof_enabled)
178 {
179   int cpu=smp_processor_id();
180
181   if (!prof_enabled)
182     return;
183
184     __set_rmid_and_cos(DISABLE_RMID,get_cos_id(cpu));
185 }
```

La función `on_switch_in_l2cat` se ejecuta cada vez que una aplicación entra a ejecutar en un núcleo y se encarga de establecer el COSID asociado a dicho núcleo en el registro `IA32_PQR_ASSOC` (salvo que el *profiling* esté deshabilitado globalmente). En este caso (particionado de caché L2) no utilizamos RMID (Resource Monitoring ID, uno de los mecanismos de monitorización de Intel RDT). Tanto RMID como COS ID se establecen escribiendo en el mismo registro *hardware*, por lo que la función `__set_rmid_and_cos` establece ambos simultáneamente.

```
187 static void on_switch_out_l2cat(pmon_prof_t* prof,
        pmcsched_thread_data_t* t, unsigned char prof_enabled)
188 {
189   if (!prof_enabled)
190     return;
191
192   __unset_rmid();
193 }
```

`on_switch_out` hace el proceso inverso. Llama a la función `__unset_rmid` que borra el valor de RMID del registro (si lo hubiera) y vuelve a colocar el COSID a 0.

```
195 static int probe_l2cat_plugin(void )
196 {
197   return !l2_cat_probe() && get_nr_coretypes()==2;
198 }
```

La función `probe_l2cat_plugin` devuelve cierto si está disponible el soporte para particionado de L2 y además existen dos tipos de cores en el sistema.

Las funciones `on_read_plugin_l2cat` y `on_write_plugin_l2cat` se encargan de imprimir la configuración actual del plugin (modo y máscaras fijadas por el usuario) y de permitir al usuario modificar tanto modo como máscaras desde /**proc**, respectivamente. No entramos en detalles.
Finalmente rellenamos la estructura `sched_ops`:

```
292 sched_ops_t l2cat_plugin = {
293   .policy                  = SCHED_L2CAT_MM,
294   .description             = "Intel L2 CAT manager",
295   .counter_config=NULL,
```

```
296     .flags                          = PMCSCHED_CPUGROUP_LOCK |
            PMCSCHED_AMP_SCHED,
297     .probe_plugin = probe_l2cat_plugin,
298     .init_plugin              = init_plugin_l2cat,
299     .destroy_plugin           = destroy_plugin_l2cat,
300     .on_read_plugin           = on_read_plugin_l2cat,
301     .on_write_plugin          = on_write_plugin_l2cat,
302     .on_switch_in_thread    =  on_switch_in_l2cat,
303     .on_switch_out_thread    =  on_switch_out_l2cat,
304 };
```

El plugin utiliza el flag `PMCSCHED_CPUGROUP_LOCK` que hace que cuando PMCSched llame a las callbacks `on_active_thread, on_inactive_thread` y `sched_periodic_timer` lo haga tras adquirir el *lock* asociado al grupo de *cores* al que pertenezca la CPU que vaya a ejecutar la rutina. Esto permite gestionar listas globales (de aplicaciones, hilos, etc) sin que las rutinas tengan que ocuparse de adquirir y liberar el *lock*. El flag `PMCSCHED_AMP_SCHED` habilita el uso de contadores de PMUs distintas en sistemas asimétricos. En este caso, no obstante, no usamos contadores *hardware* de las PMUs.

Finalmente, para que PMCSched reconozca el *plugin* es necesario declarar la constante `SCHED_L2CAT_MM` en `pmcsched.h` y añadir un puntero a `l2cat_plugin` en la lista de *plugins* disponibles.

3.4.2. Broadwell-EP

En la plataforma Broadwell-EP, la caché L3 es compartida entre los 8 *cores* de la máquina y dispone de 20 vías en total. El plugin de particionado desarrollado en este trabajo tiene una estructura muy similar al que particionaba la caché L2 en Alder Lake (descrito en la sección anterior). De igual forma comentamos aquí los aspectos más relevantes del código:

```
7  typedef enum {
8      AUTOPART_CFG_DEFAULT,
9      AUTOPART_CFG_SHARED,
10     AUTOPART_CFG_PRIVATE,
11     AUTOPART_CFG_PRIVATE_MASTER,
12     AUTOPART_CFG_USER,
13     NR_AUTOPART_CFGS
14 } autopart_mode_t;
15
16 static const char* autopart_mode_str[NR_AUTOPART_CFGS]={"default","
       shared","private","private-master","user"};
17
18 static struct {
19   autopart_mode_t autopart_mode;
20 } autopart_plugin_config;
21
22 struct user_autopart_cfg_cpu {
23   unsigned int cos_id; /* Ignored by default */
24   unsigned int mask;
25 };
```

La primera diferencia con respecto al plugin de Alder Lake es que aquí tenemos un modo extra: `private-master`. Este modo tiene el mismo funcionamiento que el modo `private`, pero da al hilo `master` de la aplicación la caché entera (lo asocia al COS 0). Esto permite evitar que los hilos `master` de algunas aplicaciones reciban una partición de caché demasiado pequeña al principio de su ejecución, cuando aún no se han creado los demás hilos. Además, puesto que 20 no es divisible entre 8 en los enteros, en esta máquina el modo `private` divide la caché L3 en particiones disjuntas otorgando 2 vías de caché a cada una de 4 particiones y otras 3 vías a cada una de otras 4 particiones (en total, se reparten las 20 vías).

A continuación presentamos la configuración de contadores del módulo, puesto que en este caso vamos a exportar algunos contadores virtuales. Esto se debe a que Intel RDT ofrece también la tecnología Intel CMT (*Cache Monitoring Technology*) la cual ofrece capacidades de monitorización de la caché adicionales a los contadores *hardware*. Esta información la vamos a exportar a través de los contadores virtuales del módulo:

```
30 static pmcsched_counter_config_t cconfig = {
31   .counter_usage = {
32     .hwpmc_mask = 0x0, /*  bitops -h 0,1*/
33     .nr_virtual_counters = CMT_MAX_EVENTS,
34     .nr_experiments = 0,
35     .vcounter_desc = {"llc_usage", "total_llc_bw", "local_llc_bw"},
36   },
37   .pmcs_descr = NULL,
38   .metric_descr = {NULL, NULL},
39   .profiling_mode = TBS_SCHED_MODE, /* Not needed really */
40 };
```

- `counter_usage` nos permite definir los contadores hardware y virtuales que exporta el módulo:
  - `hwpmc_mask`: Máscara que define los contadores hardware "forzados", es decir, aquellos que exporta el módulo. En nuestro caso ninguno, puesto que sólo vamos a exportar contadores virtuales.
  - `nr_virtual_counters`: Número de contadores virtuales que exporta el módulo. En nuestro caso son `CMT_MAX_EVENTS` (que vale 3) puesto que vamos a exportar los eventos CMT (los distintos tipos de datos que ofrece Intel CMT).
  - `nr_experiments`: Multiplexación de eventos (distintos conjuntos de eventos hardware se monitorizan siguiendo un algoritmo Round Robin, en el caso de que tengamos más eventos que monitorizar que contadores hardware). En nuestro caso 0 puesto que no utilizamos contadores hardware.
  - `vcounter_desc`: Descripción de alto nivel (cadena de caracteres) de los `nr_virtual_counters` contadores virtuales.
- `pmcs_descr`: Descripción de los contadores hardware que el API de PMCTrack pueda utilizar. Ninguno en nuestro caso, puesto que no utilizamos contadores hardware.

- `metric_descr`: Descripción de métricas a calcular con los valores de los contadores hardware. De nuevo, nada en nuestro caso.
- `profiling_mode`: Define cada cuanto se obtiene una muestra de los contadores. Admite TBS y EBS, como la herramienta `pmctrack`. En este caso, al no haber configurado contadores hardware, se ignora.

```
41 /* WARNING: HOBBES SPECIFIC */
42 #define MAX_AUTOPART_CPUS 8
43 #define MAX_AUTOPART_CORES_IN_GROUP 8
44 struct user_autopart_cfg_cpu user_autopart_cfg[MAX_AUTOPART_CPUS];
```

A continuación se definen las macros `MAX_AUTOPART_CPUS` y `MAX_AUTOPART_CORES_IN_GROUP` que permiten adaptar fácilmente el plugin para otras máquinas con diferente topología. La segunda indica el número de CPUs que hay en cada *socket* de la máquina (en nuestro caso están todos en un único *socket*).

`update_llc_configuration_cpu`, `update_llc_configuration` y `get_cos_id` son prácticamente idénticas al plugin de Alder Lake. La macro `EBIT(i)` devuelve una máscara con el bit `i` a 1 y el resto a 0. Las máscaras para el modo `private` quedan de la siguiente manera:

```
45     unsigned int private_mask_core_ccx[]={ EBIT(1) | EBIT(2) | EBIT(3),
46                                            EBIT(4) | EBIT(5) | EBIT(6),
47                                            EBIT(7) | EBIT(8) | EBIT(9),
48                                            EBIT(10) | EBIT(11) | EBIT
                                                 (12),
49                                            EBIT(13) | EBIT(14),
50                                            EBIT(15) | EBIT(16),
51                                            EBIT(17) | EBIT(18),
52                                            EBIT(19) | EBIT(20) };
```

```
111 static int init_intel_plugin_autopart(void)
112 {
113     int i;
114
115   autopart_plugin_config.autopart_mode=AUTOPART_CFG_DEFAULT;
116
117     trace_printk("Loading LLC CAT Manager\nnr_cos=%d\nmask=0x%x\n",
118             pmcs_cat_support.cat_nr_cos_available,
119             pmcs_cat_support.cat_cbm_mask);
120
121     /* Initialize user settings */
122     for (i=0;i<num_active_cpus();i++){
123         struct user_autopart_cfg_cpu* user_cfg=&user_autopart_cfg[i];
124         user_cfg->cos_id=(i%MAX_AUTOPART_CORES_IN_GROUP)+1;
125         user_cfg->mask=pmcs_cat_support.cat_cbm_mask;
126     }
127
128     update_llc_configuration();
129
130   return 0;
131 }
132
```

```
133 static void destroy_intel_plugin_autopart (void)
134 {
135     /* Restore defaults */
136     autopart_plugin_config.autopart_mode=AUTOPART_CFG_DEFAULT;
137     update_llc_configuration();
138 }
```

En las funciones para inicializar y destruir el plugin el mayor cambio respecto al plugin de Alder Lake es que no tenemos que encargarnos de inicializar ni destruir el soporte global de particionado Intel CAT *L3*. Lo hace PMCSched automáticamente.

```
139 static void on_switch_in_intel_autopart(pmon_prof_t* prof,
    pmcsched_thread_data_t* t, unsigned char prof_enabled)
140 {
141   int cpu=smp_processor_id();
142     struct task_struct* p=prof->this_tsk;
143
144   if (!prof_enabled)
145     return;
146
147     /* Just establish the cos-id in the per-thread field */
148     if (autopart_plugin_config.autopart_mode==
        AUTOPART_CFG_PRIVATE_MASTER
149             && p->pid==p->tgid)
150         t->thread_cos=0; /* Master thread uses full LLC */
151     else
152         t->thread_cos=get_cos_id(cpu);
153 }
154
155 static void on_switch_out_intel_autopart(pmon_prof_t* prof,
    pmcsched_thread_data_t* t, unsigned char prof_enabled)
156 {
157     /* Nothing to do, because PMCsched already clears the stuff */
158 }
```

En `on_switch_in_intel_autopart` tenemos que comprobar (cuando el modo es `private-master`) si el hilo que va a entrar en la CPU es el hilo principal de la aplicación. En tal caso le asignamos el COS 0 y en caso contrario el COS ID que toque según el modo y la CPU (devuelto por `get_cos_id`). Nótese que en este caso para asignar el COS ID estamos simplemente escribiéndolo en el campo `thread_cos`, mientras que en el Alder Lake teníamos que escribir directamente el RMID y el COS al registro correspondiente. Esto se debe a que, para el caso de caché L3, PMCSched se encarga (en sus funciones `pmcsched_on_switch_in` y `pmcsched_on_switch_out`) de escribir y limpiar el registro correspondiente, por lo que no tenemos que hacerlo nosotros. PMCSched sí que coloca en este caso un valor de RMID distinto de 0, ya que es necesario para poder utilizar las tecnologías CMT (*Cache Monitoring Technology*) y MBM (*Memory Bandwidth Monitoring*), que en este *plugin* necesitamos para obtener información del ancho de banda o espacio total de la caché consumido por la aplicación, que luego exportamos mediante contadores virtuales.

Para que el *plugin* pueda cargarse necesitamos no sólo el soporte hardware (Intel RDT) y estar en un sistema simétrico (`get_nr_coretypes() == 1`) sino que el

número de COS ID disponibles sea al menos el número de CPUs +1, ya que necesitamos un COS por CPU $(1, 2, ...)$ y otro para el hilo principal (COS 0) en el modo `private-master`. En el caso de nuestra máquina esto significa que necesitamos mínimo $8 + 1 = 9$ IDs. Como hay disponibles 16 esto no es un problema. La función `probe_intel_autopart_plugin` se encarga de realizar las comprobaciones.

```
169 static int on_new_sample_autopart(pmon_prof_t *prof,
170     int cpu, pmc_sample_t *sample, int flags, void *data)
171 {
172   sched_thread_group_t* cur_group=get_cur_group_sched();
173     pmcsched_thread_data_t *t = prof->monitoring_mod_priv_data;
174   app_t_pmcsched* app;
175   app_t* app_cache;
176   int i;
177   int cnt_virt=0;
178
179   if (t==NULL)
180     return 0;
181
182     app=get_group_app_cpu(t,cur_group->cpu_group->group_id);
183     app_cache=&app->app_cache;
184
185   /* Copy RMID and COS ID to per-thread structure to maintain per-
        thread info */
186   t->cmt_monitoring_data.rmid = app_cache->app_cmt_data.rmid;
187   t->cmt_monitoring_data.cos_id = t->thread_cos;
188   intel_cmt_update_supported_events(&pmcs_cmt_support,&t->
        cmt_monitoring_data);
189
190   /* Embed virtual counter information so that the user can see what's
        going on */
191   for(i=0; i<CMT_MAX_EVENTS; i++) {
192     if ((prof->virt_counter_mask & (1<<i) )) {
193       sample->virt_mask|=(1<<i);
194       sample->nr_virt_counts++;
195       sample->virtual_counts[cnt_virt]=t->cmt_monitoring_data.
            last_llc_utilization[i];
196       cnt_virt++;
197     }
198   }
199
200   return 0;
201 }
```

La rutina `on_new_sample_autopart` se encarga de exportar las muestras de los contadores virtuales y se llama cada vez que se recoge una muestra de los contadores hardware. Recibe los valores de estos (si estuviésemos utilizando contadores hardware, que no lo hacemos en este *plugin*) en la estructura apuntada por `sample` y debe rellenar ahí los valores de los contadores que exporta el módulo, además de hacer cualquier procesamiento adicional (almacenar los valores de los contadores en alguna estructura interna del módulo, por ejemplo).

En nuestro caso, lo que hace la función es calcular los eventos que ofrece Intel CMT (que se corresponden con los contadores que queremos exportar), y luego relle-

na los valores obtenidos en `sample`. `prof->virt_counter_mask` contiene la máscara de contadores virtuales que se deben exportar y `sample->virt_mask` la máscara de contadores virtuales que están presentes en `sample`.

Finalmente rellenamos la estructura `sched_ops`. Esta vez hemos rellenado el puntero `counter_config` e incluido el `flag` `PMCSCHED_PER_THREAD_QOS_HANDLING` que provoca que PMCSched realice el particionado a nivel de hilo (toma el valor del COS ID del campo `thread_cos` de cada hilo en vez de utilizar un valor de COS ID global para todos los hilos de la aplicación):

```
289 sched_ops_t intel_autopart_plugin = {
290   .policy                    = SCHED_INTEL_AUTOPART_MM,
291   .description               = "LLC CAT manager (Intel version)",
292   .counter_config = &cconfig,
293   .flags                       = PMCSCHED_CPUGROUP_LOCK |
        PMCSCHED_PER_THREAD_QOS_HANDLING,
294   .probe_plugin = probe_intel_autopart_plugin,
295   .init_plugin             = init_intel_plugin_autopart,
296   .destroy_plugin          = destroy_intel_plugin_autopart,
297   .on_read_plugin          = on_read_intel_plugin_autopart,
298   .on_write_plugin         = on_write_intel_plugin_autopart,
299   .on_switch_in_thread   =  on_switch_in_intel_autopart,
300   .on_switch_out_thread    =  on_switch_out_intel_autopart,
301 };
```

Capítulo 4

Análisis Experimental

En este capítulo se evalua la efectividad derivada de particionar la caché utilizando los *plugins* presentados en capítulos anteriores. La política de particionado consistirá en dividir la caché compartida creando particiones equitativas y disjuntas que se asignan de forma fija a cada *core*. Se corresponde con el modo `private` de ambos *plugins*. Para evaluar la idoneidad del particionado se compara el rendimiento de varios *benchmarks* cuando la caché está compartida frente a cuando está particionada.

El objetivo de este particionado es identificar aplicaciones que se beneficien de este particionado para estudiar por qué lo hacen y finalmente tratar de proponer políticas de particionado más complejas que puedan ser beneficiosas.

4.1. Benchmarks

Hemos elegido una serie de *benchmarks* multihilo que permiten personalizar (mediante el paso de algún parámetro) el número total de hilos de la aplicación. En nuestros experimentos las aplicaciones se ejecutan con tantos hilos como CPUs (*cores*) haya en el sistema.

Disponemos de *benchmarks* HPC (*High Performance Computing*) que consisten en aplicaciones que resuelven problemas científicos (cálculo de soluciones de ecuaciones diferenciales, factorización de matrices, etc) de forma muy rápida empleando múltiples hilos que trabajan en paralelo. Esto incluyó inicialmente *benchmarks* de las *suites* NPB (*NAS Parallel Benchmarks*) [27], PARSEC [28], Rodinia [29] y el *benchmark* RNASeq, una aplicación de bioinformática que se encarga de predecir si dos hebras de ácido nucleico interactúan [30]. Posteriormente también incluimos las *suites* BOTS [31] y TP-Parsec [32]. TP-Parsec es una reimplementación de la suite PARSEC realizada de forma independiente, que explota paralelismo de tareas. Esta aproximación de paralelización produce *benchmarks* cuyos hilos presentan un comportamiento más heterogéneo que en el caso de PARSEC, lo que como veremos a lo largo del capítulo va a ser relevante de cara al particionado. También hemos añadido soporte para ejecutar los *benchmarks* PBBS (*Problem Based Benchmark*

Suite [17]) en Het-Harness. Finalmente utilizamos algunos *benchmarks* de Cloudsuite [33] (que no son HPC sino Cloud), aprovechando el soporte para contenedores en Het-Harness, recientemente añadido.

En definitiva, en este estudio se emplea un número bastante elevado de *benchmarks* que habitualmente se utilizan en investigación para evaluar el rendimiento de sistemas multiprocesador.

A lo largo de este capítulo vamos a referirnos a los *benchmarks*, por comodidad, mediante su nombre en el *script* de Het-Harness. De esta forma nos referiremos al *benchmark* `LU` de la *suite* NPB como `lu_npb`, salvo en el caso de la *suite* PBBS, donde obviaremos el subfijo `_pbbs` para evitar que los nombres de los *benchmarks* sean demasiado largos.

4.2. Particionado

Recordamos que en el caso de la plataforma Alder Lake solo vamos a particionar las cachés L2 de los *sockets* de E-*cores* (recordamos que los P-*cores* disponen cada uno de una caché L2 privada), y que particionamos la caché de forma equitativa (cada uno de los 4 núcleos del *socket* recibe 4 de las 16 vías de la caché L2). Llamamos particionado estático a esta política de particionado de caché (por hilo, pues fijamos cada hilo a un *core*). Hay que tener en cuenta que existe una caché L3 compartida por todos los núcleos del sistema, lo que limita a priori el impacto que el particionado de la caché L2 pueda tener. En la figura 4.1 se muestra la topología de la plataforma, junto con las cachés L2 compartidas divididas en vías. Esto permite visualizar gráficamente el efecto del particionado estático en la caché.

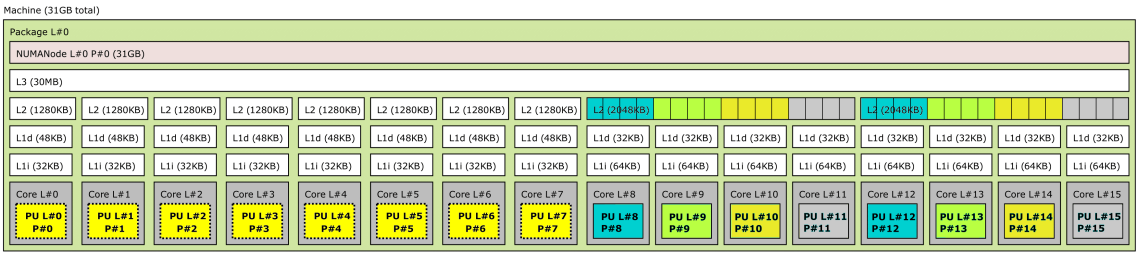

Figura 4.1: Caché L2 particionada en los *sockets* de E-*cores* de nuestra plataforma Alder Lake. Se muestran las cachés L2 de los E-*cores* divididas por vías, con las vías que obtiene cada núcleo marcadas con el mismo color que dicho núcleo.

Por otro lado, en la plataforma Broadwell-EP particionamos el último nivel de caché (L3), que es compartido por todos los núcleos del sistema, por lo que esperamos que tenga más impacto el particionado. Disponemos de 20 vías y 8 *cores*. Se asignan 3 vías a los 4 primeros núcleos y 2 a los 4 últimos. En la figura 4.2 se muestra la topología de la plataforma junto con el efecto del particionado (nótese como los primeros núcleos reciben 3 vías de caché y los últimos solo 2).

Finalmente, recordamos que el particionado a nivel de hilo solo afecta al reemplazo, porque todos los hilos de una misma aplicacion comparten memoria por lo que un hilo puede acceder y tener una acierto de caché de una línea que ha traído otro hilo a la caché.

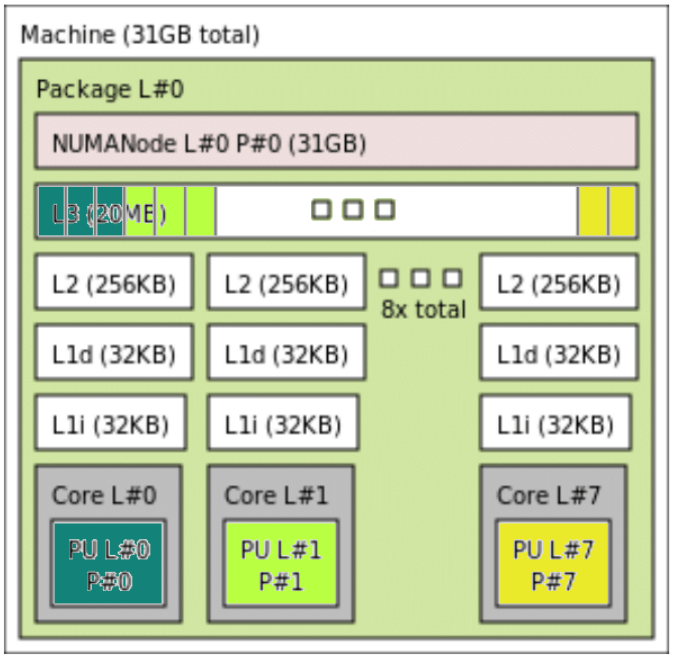

Figura 4.2: Caché L3 particionada en los *sockets* de E-*cores* de nuestra plataforma Broadwell-EP. Se muestran la caché L3 dividida por vías, con las vías que obtiene cada núcleo marcadas con el mismo color que dicho núcleo.

4.3. Alder Lake

Comenzamos nuestro análisis en la plataforma Alder Lake. Como se describió en el capítulo 3, esta plataforma dispone de 16 núcleos en total y una arquitectura asimétrica en la que hay P-*cores* (procesadores rápidos) y E-*cores* (procesadores lentos pero más eficientes en consumo de energía). Nuestro sistema Alder Lake tiene 8 de ambos. Cada procesador dispone de una caché L1 privada. La caché L2 es privada en el caso de los P-*cores* (con 1,25 MB de capacidad y 11 vías). En el caso de los E-*cores* cada 4 procesadores comparten una caché L2 de 2 MB de capacidad (y 16 vías). Esta es la caché que el hardware nos permite particionar. También existe una caché L3 de 30 MB de capacidad compartida entre los 16 procesadores.

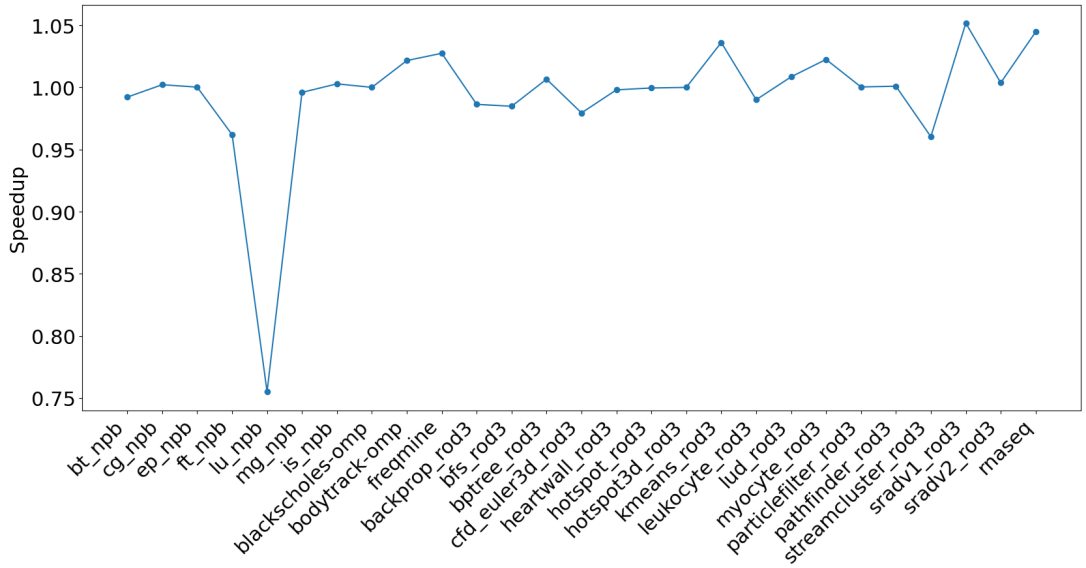

Figura 4.3: Rendimiento (speedup) de diversos *benchmarks* al particionar la caché de forma estática.

La figura 4.3 muestra el beneficio relativo (*speedup*) al particionar estáticamente algunos *benchmarks*. Más formalmente se define *speedup* como:

$$\text{speedup} = \frac{\text{Tiempo de ejecución con caché compartida}}{\text{Tiempo de ejecución con particionado estático}}$$

De esta forma, valores *speedup* > 1 indican mejoría de rendimiento bajo el particionado, mientras valores menores que 1 indican degradación de rendimiento.

En general, vamos a considerar que menos de un 5 % de variación (es decir, valores de *speedup* entre 0,95 y 1,05) no es significativo. Como podemos observar el efecto de nuestra política de particionado estático no es muy dramático: Algunos de los benchmarks empeoran levemente su rendimiento mientras otros lo mejoran levemente. La única excepción a esto es el benchmark `lu_npb` que empeora su rendimiento casi un 25 %. Dicho benchmark se ejecuta con el conjunto de datos de entrada C, mientras que `bt_npb`, que es otro resolvedor de ecuaciones similar se ejecuta con el conjunto de datos B (que es el mayor conjunto de datos disponible para este *benchmark*). Los tamaños de los conjuntos de datos se especifican en [34], pero esencialmente entre los conjuntos A, B, C cada uno multiplica por 4 el tamaño del conjunto anterior, siendo C el más grande. Esto nos lleva a plantearnos si es el tamaño del conjunto de datos el causante de la degradación de rendimiento:

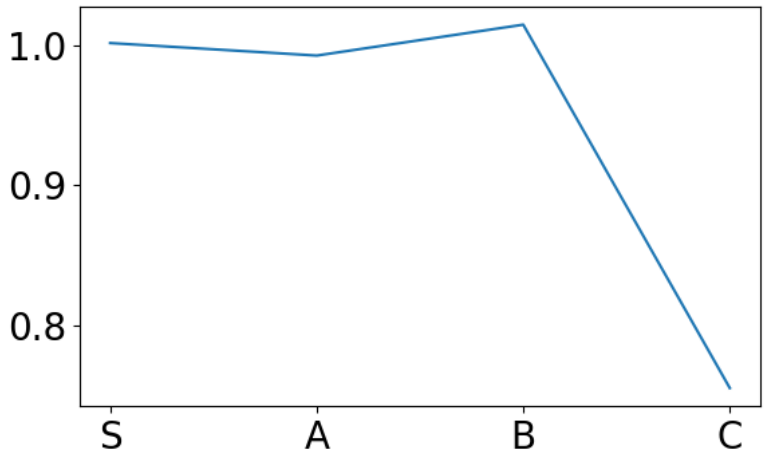

Figura 4.4: Speedup de `lu_npb` en función del conjunto de datos de entrada.

La figura 4.4 muestra los valores de *speedup* de `lu_npb` cuando se ejecuta con S, A, B o C como conjunto de datos de entrada. La figura 4.4 muestra que, en efecto, los conjuntos de datos más pequeños no muestran una pérdida de rendimiento reseñable. Procedemos a ejecutar todos los benchmarks de NAS NPB para los conjuntos de datos A, B, C (si dicho conjunto de datos está disponible para un *benchmark* dado):

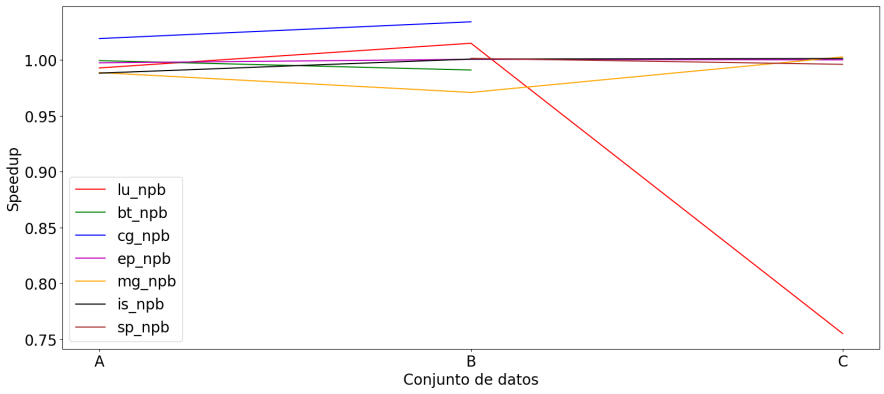

Figura 4.5: Speedup en función del conjunto de datos de entrada, para todos los *benchmarks* NPB.

La figura 4.5 muestra el *speedup* para todos los *benchmarks* NPB en función del conjunto de datos de entrada. Como podemos ver, no se observa una pérdida de rendimiento al aumentar el tamaño de los datos en el resto de los benchmarks (salvo en el ya mencionado `lu_npb`). Por tanto, entendemos que la pérdida de rendimiento en

ese caso es debida a que el conjunto de trabajo (*working set*) C no cabe en la caché L2 cuando esta no está compartida por todos los hilos, mientras que los de tamaño S, A y B sí caben. Asumiendo que el conjunto C cabe en la caché $L2$ cuando esta está compartida (pues el rendimiento es mucho mejor), conjeturamos que particionar la caché está provocando (paradójicamente) que los hilos "se pisen" unos a otros, pues la caché está compartida para lectura pero particionada para reemplazar líneas de caché. Es decir, aunque una cierta línea esté siendo reutilizada continuamente por los distintos hilos de la aplicación podría ser sustituida por el hilo "propietario" de la línea aunque existan en la caché otras líneas menos utilizadas, pero que se encuentran en otras particiones por lo que no pueden ser reemplazadas. Al ser una línea muy utilizada es de esperar que un hilo la vuelva a traer rápidamente a la caché, desplazando en el caso peor de nuevo a otra línea muy usada. A esto nos referimos cuando decimos que los hilos "se pisan": Se acaban produciendo muchos reemplazos de líneas muy usadas que no deberían ser reemplazadas mientras otras poco usadas se mantienen.

Como veremos más adelante las aplicaciones homogéneas (todos los hilos hacen en paralelo el mismo trabajo) como `lu_npb` van a ser peores candidatas para beneficiarse de este particionado que aplicaciones más heterogéneas.

En el caso del resto de *benchmarks* de NPB su comportamiento distinto puede deberse a que el patrón de acceso a caché sea distinto, puesto que cada *benchmark* realiza una factorización sobre matrices distinta y es de esperar que el patrón resultante de aplicar una factorización LU no sea el mismo que en otras formas de factorizar.

A continuación analizamos los resultados de los benchmarks de PARSEC: `blackscholes`, `bodytrack` y `freqmine`. `freqmine` se caracteriza por tener un working set enorme (en torno a 1 GB) y una necesidad de capacidad de caché voraz mientras que los de blackscholes y bodytrack caben en la caché (de último nivel) [28].

Como hemos visto en la figura 4.3 no hay diferencia significativa entre particionar o no. Esto se debe a que, como se muestra en [28], estas aplicaciones tienen una tasa de fallos (*miss rate*) de aproximadamente 0 % cuando se ejecutan en una caché compartida de 4 vías y una capacidad de 2 MB. El miss rate apenas aumenta para una caché del mismo número de vías y una capacidad de 512 kB, que es la partición de caché que toca a cada de uno de nuestros E-cores en el particionado, teniendo en cuenta que además pueden acceder a toda la caché en lectura (el particionado solo se aplica en reemplazo).

Los benchmarks de Rodinia ofrecen también casi todos un rendimiento muy similar al particionar que al no hacerlo. Como los resultados que hemos obtenido no son muy prometedores procedemos a ejecutar también *benchmarks* de las *suites* bots y Parsec-TP. En la figura 4.6 se muestra el *speedup* obtenido en cada uno de ellos.

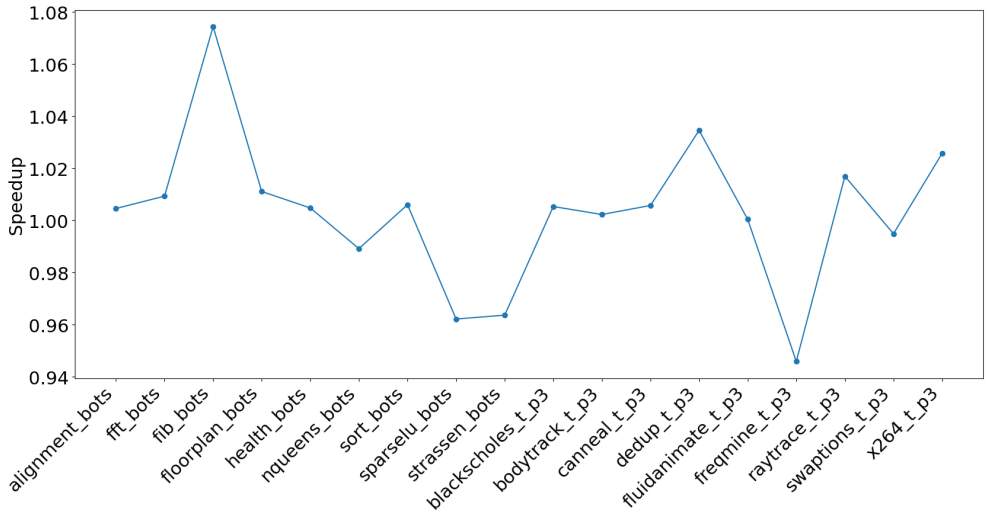

Figura 4.6: Speedup de los *benchmarks* bots y Parsec-TP.

Como podemos ver en la figura 4.6 sólo un benchmark muestra una mejora significativa (*speedup* superior a 1,05). En este punto se compilan los *benchmarks* de la *suite* PBBS [17] y se han elaborado los *scripts* de `bash` para automatizar el lanzamiento de estos *benchmarks* con la herramienta Het-Harness. El resto de *benchmarks* que hemos usado hasta ahora ya estaban compilados en la máquina.

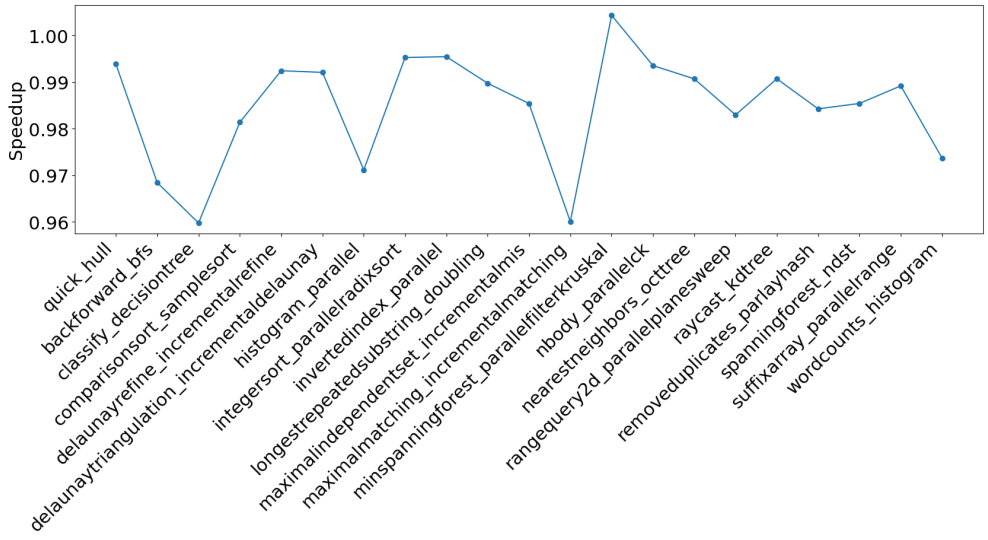

Figura 4.7: Speedup de los *benchmarks* de la suite PBBS.

Como se ve en la figura 4.7, en este caso ninguno mejora significativamente aunque tampoco vemos ninguno que empeore significativamente. Con estos resultados en la mano concluimos que particionar la caché $L2$ entre hilos no parece un camino muy prometedor. Hay que tener en cuenta que los 8 *cores* más rápidos disponen de cachés L2 privadas y por tanto el efecto de nuestro particionado va a estar limitado

pues sólo podemos particionar las cachés L2 de los 8 E-*cores*. Procedemos a cambiar de máquina e intentar la misma estrategia pero particionando la caché $L3$.

4.4. Broadwell-EP

En la plataforma Broadwell-EP vamos a repetir los experimentos de la sección anterior (mismos *benchmarks*), pero particionando la caché L3. Como adelantábamos en el capítulo anterior en esta plataforma tenemos una caché de último nivel $L3$ con 20 vías compartidas por los 8 *cores* del sistema (y 20 MB de capacidad). Al particionar la caché el *plugin* de particionado otorga a 4 *cores* 2 vías a cada uno y 3 vías a cada uno de los otros 4.

Los experimentos que vamos a realizar son los mismos que en la plataforma Alder Lake. Empezamos con la discusión de los resultados de los programas de NPB, Parsec (`bodytrack`, `blackscholes` y `freqmine`), Rodinia y `RNASeq`.

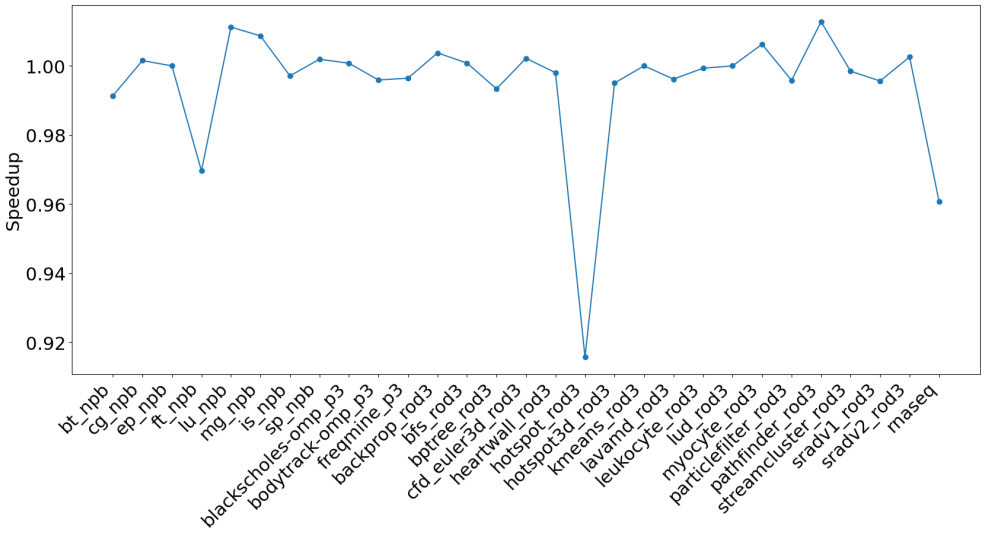

Figura 4.8: Speedup de los *benchmarks* de varias *suites*: NPB, Parsec, Rodinia y `RNASeq`.

La figura 4.8 muestra el *speedup* proporcionado por el particionado frente a caché compartida para los *benchmarks* arriba mencionados. En la figura observamos que sólo `hotspot_rod3` empeora significativamente. Este es un *benchmark* que resuelve de forma iterativa ecuaciones diferenciales para estimar la temperatura de un procesador (2D) [29]. Por otro lado, la versión 3D (`hotspot3d_rod3`) no muestra una degradación significativa del rendimiento.

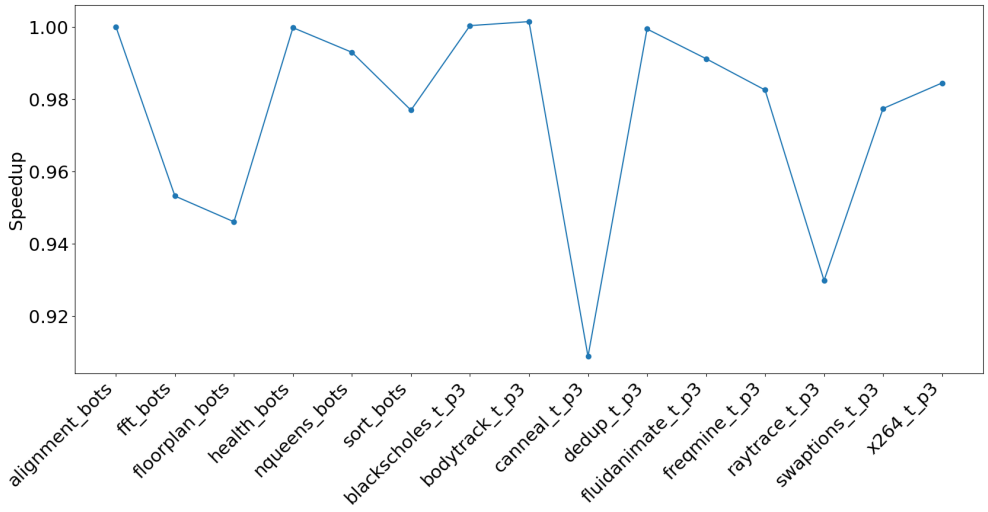

Figura 4.9: Speedup de los *benchmarks* de la *suite* bots y Parsec-TP.

La figura 4.9 muestra los resultados del particionado para aplicaciones de bots y Parsec-TP. Vemos que unos cuantos que empeoran significativamente (`floorplan_bots`, `canneal_t_p3` y `raytrace_t_p3`), aunque una gran parte de los *benchmarks* siguen sin mostrar diferencias significativas entre particionar o no.

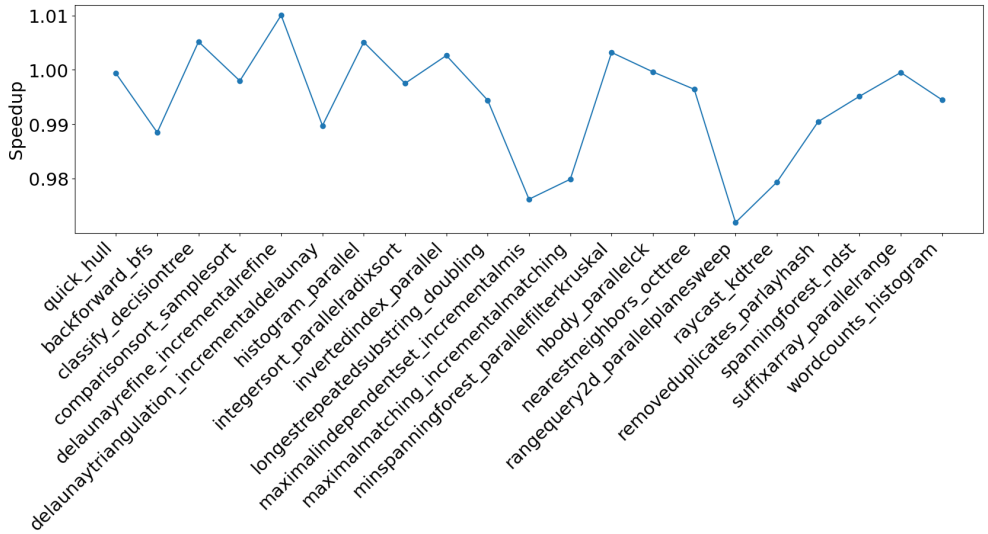

Figura 4.10: Speedup de los *benchmarks* de la *suite* PBBS.

Como vemos en la figura 4.10, que muestra el *speedup* de los *benchmarks* de PBBS bajo el particionado de caché L3, ningún *benchmark* muestra diferencia al particionar. En total hemos ejecutado 64 *benchmarks* de los cuáles ninguno mejora con particionado estático y solo 4 empeoran significativamente. Son resultados muy poco prometedores que no obstante vamos a analizar para intentar comprender por qué algunas aplicaciones sí que empeoran bajo el particionado, puesto que hemos

experimentado con un conjunto muy diverso de *benchmarks*, sin éxito.

Para llevar a cabo el análisis pormenorizado de los resultados se utilizó HetHarness y el soporte de contadores *hardware* de la herramienta PMCTrack (y los contadores virtuales que exportamos en nuestro plugin de particionado) para recopilar información de contadores durante la ejecución de nuestros *benchmarks* (con la caché $L3$ compartida por todos los hilos). Recopilamos la información por hilo y aplicamos una serie de métricas para poder interpretar de forma intuitiva a los datos obtenidos de los contadores.

4.4.1. Métricas

De cara a realizar un análisis de los datos vamos a utilizar las siguientes métricas:

- `llcrpkc` (*LLC Requests per Kilo Cycle*): Como el nombre indica mide el número de accesos a la caché de último nivel por cada 1000 ciclos del procesador.
- `llcmpkc` (*LLC Misses per Kilo Cycle*): En este caso lo que se mide es el número de fallos de caché de último nivel por cada 1000 ciclos del procesador.
- `l1bw`: Ancho de banda de la caché L1, medido como fallos de caché L1 por cada 1000 instrucciones, ya que los fallos de caché provocan reemplazos de líneas de caché. Por tanto, a mayor número de fallos de caché, mayor es el tamaño de los datos a transferir a la caché L1. Es decir, mayor ancho de banda.
- `l2bw`: Ancho de banda de la caché L2, medido de nuevo como fallos de caché L2 por cada 1000 instrucciones.
- `stalls_l2_miss`: Número de ciclos de parada del procesor debido a un fallo de caché L2.
- `llcm_bandwidth`: Ancho de banda de la caché L3, medido como número de fallos de caché de último nivel multiplicado por 64 y dividido entre el tiempo empleado (es decir, lo calcula en base al número de transferencias de memoria a caché $L3$).
- `l1reuse`: Fallos de L1 en relación a fallos de L2. Un valor alto indica que la caché L1 hace mucho reuso de los datos contenidos en la caché L2.
- `l2reuse`: Fallos de L2 en relación a fallos de L3.

4.4.2. Análisis de datos (por aplicación)

Las distintas aplicaciones se pueden clasificar en distintas categorías según según su comportamiento en el acceso a la caché y a la memoria, para luego tomar decisiones de particionado en base a la clase asignada. Una clasificación posible es la que se presenta en [12], que divide las aplicaciones en tres clases disjuntas:

- *Cache-sensitive*: Aquellos programas que sufren una importante degradación del rendimiento cuando se les asignan menos vías de caché, inferiores a un cierto umbral específico de la aplicación.
- *Streaming*: Programas cuyo rendimiento casi no depende del número de vías asignado. Es decir, no muestran degradación del rendimiento cuando se reduce su espacio de caché (no aprovechan la caché, principalmente por motivos de localidad).
- *Light-sharing*: Programas que no son ni *cache-sensitive* ni *streaming*. Estos suelen ser programas en los que el conjunto de trabajo cabe en los niveles de caché privados inmediatamente superiores a la caché compartida.

La principal diferencia entre las clases *streaming* y *light-sharing* es que las aplicaciones de la primera dañan el rendimiento de otras aplicaciones que se ejecutan simultáneamente, mientras que las aplicaciones *light-sharing* no lo hacen. Por ejemplo, si el conjunto de trabajo cabe en la caché idealmente no se harán muchas peticiones a memoria, con lo que la aplicación casi no reemplazará líneas.

Nótese que la clase de una aplicación depende de la caché de la que dispone la máquina. Por ejemplo, algunas aplicaciones *streaming* pueden ser *cache-sensitive* en una máquina con una caché con más capacidad. Y al revés, aplicaciones *cache-sensitive* pueden pasar a ser *streaming* si la caché es más pequeña. Aunque en nuestro caso estamos buscando particionar hilos de aplicaciones cuando son las únicas en ejecución en el sistema, vamos a tratar de obtener una clasificación similar, como punto de partida en nuestro análisis de datos. Por tanto, vamos a hacer en ocasiones referencia a conceptos asociados con esta clasificación (por ejemplo, al hablar de "comportamiento *streaming*"), aunque la naturaleza de las aplicaciones sea diferente.

De cara a realizar nuestro análisis comenzamos agregando la información por aplicación (es decir, vamos a trabajar inicialmente sobre el comportamiento global de la aplicación, que consiste en calcular la media de cada métrica) y buscamos correlaciones con el *speedup* (coeficiente de correlación de Pearson):

llcrpkc	llcmpkc	llcm_bandwith	l1bw	l2bw	l1reuse	l2reuse
-0.075102	0.047291	0.045769	-0.148971	-0.030132	-0.079322	-0.036342

Estas correlaciones no nos dan ninguna información, puesto que todos los coeficientes están demasiado alejados de 1 y -1 y muy cercanos a 0. Vamos a elegir la intensidad en memoria como la variable con la que empezar a trabajar, para intentar separar las aplicaciones en las que las métricas de reuso `l1reuse` y `l2reuse` son relevantes de las que no. De hecho, en la clasificación que acabamos de presentar la clase *light-sharing* contiene necesariamente aplicaciones poco intensivas en memoria, mientras que la clase *streaming* contiene aplicaciones intensivas en memoria. Además, al menos intuitivamente parece claro que una aplicación que no es intensiva en memoria (accede poco a la memoria principal) es una aplicación que emplea una fracción muy reducida de su tiempo de ejecución llevando información a la caché, y

por tanto no debería mostrar mejorías significativas de rendimiento entre particionar y no particionar. Y en tal caso nos interesa identificar estas aplicaciones y separarlas del resto.

Puesto que lo primero (según esta clasificación) va a ser clasificar los *benchmarks* en intensivos en memoria y no intensivos en memoria acudimos a la métrica `llcm_bandwith` que nos da el ancho de banda de la caché L3 (calculado en función de los fallos de caché) y buscamos un umbral para separar las aplicaciones en intensivas en memoria y no intensivas en memoria.

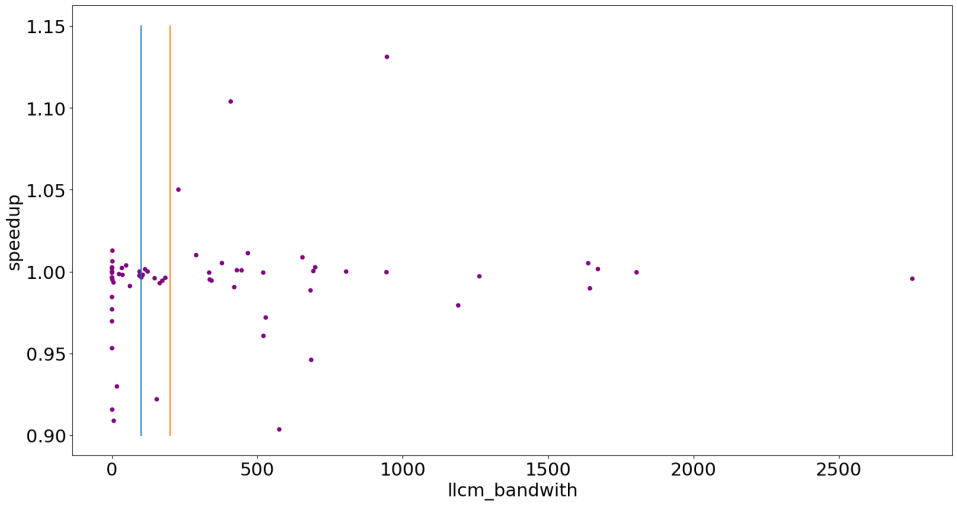

Figura 4.11: La línea azul marca `llcm_bandwidth=100` y la naranja `llcm_bandwidth=200`.

La figura 4.11 muestra el *speedup* en función del ancho de banda de caché L3 (`llcm_bandwidth`), con las líneas azul y naranja mostrando potenciales umbrales para separar *benchmarks* intensivos en memoria de no intensivos (las líneas marcan `llcm_bandwidth = 100` y `llcm_bandwidth = 200`, respectivamente). Elegimos el punto de corte `llcm_bandwidth=100`, es decir, consideramos no intensivos en memoria a aquellos con `llcm_bandwith < 100`. Esto nos hace sospechar que los miembros de esta clase son aquellos cuyo conjunto de trabajo cabe en la caché L3 (o incluso en la cachés L1 y L2, como en el caso de `swaptions_t_p3`). Esto queda confirmado por el baja ratio $\frac{llcmpkc}{llcrpkc}$ (de media apenas un 6 % en esta clase, frente al 39 % de los *benchmarks* con `llcm_bandwith > 100` y los valores más altos de `l1reuse` y/o `l2reuse`).

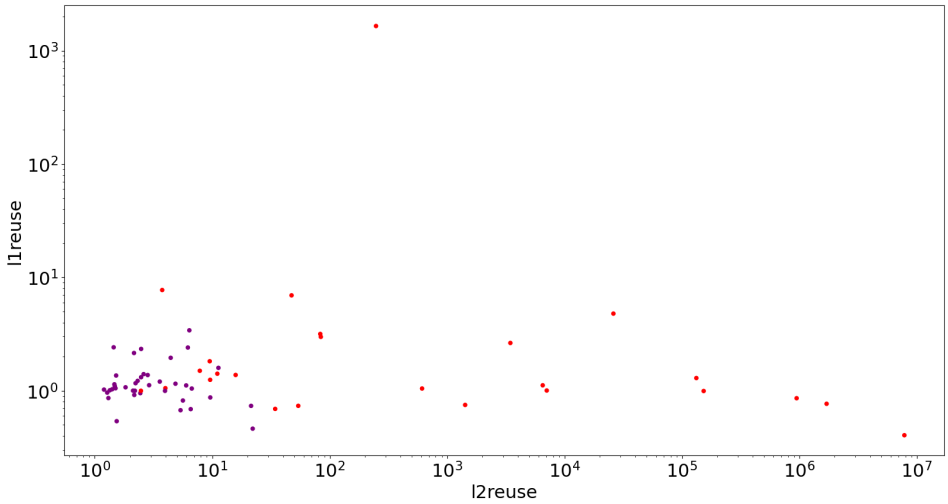

Figura 4.12: En rojo, los *benchmarks* no intensivos en memoria. Gráfico en escala logarítmica.

En la figura 4.12 podemos ver la distribución de los *benchmarks* en los ejes (`l1reuse`, `l2reuse`), con aquellos que tienen `llcm_bandwidth < 100` marcados con color rojo. Hemos elegido `100` como punto de corte (aunque como hemos visto hasta `200` parece razonable) puesto que en cuanto aumentamos ligeramente (`113`) este valor empiezan a aparecer algunos *benchmarks* como `blackscholes_t_p3` que presentan un comportamiento más *streaming* (valores muy bajos de `l1reuse` y `l2reuse` y alto porcentaje de fallos de L3).

Como podemos ver en la figura 4.12 las aplicaciones con `llcm_bandwidth < 100` presentan valores mucho más altos de `l1reuse` y `l2reuse` que aquellas con `llcm_bandwidth > 100`, dejando a estas en la esquina del gráfico (visualmente se aprecia una división bastante clara). Un vistazo al mismo gráfico realizado usando `llcm_bandwidth=200` confirma esta idea (figura 4.13).

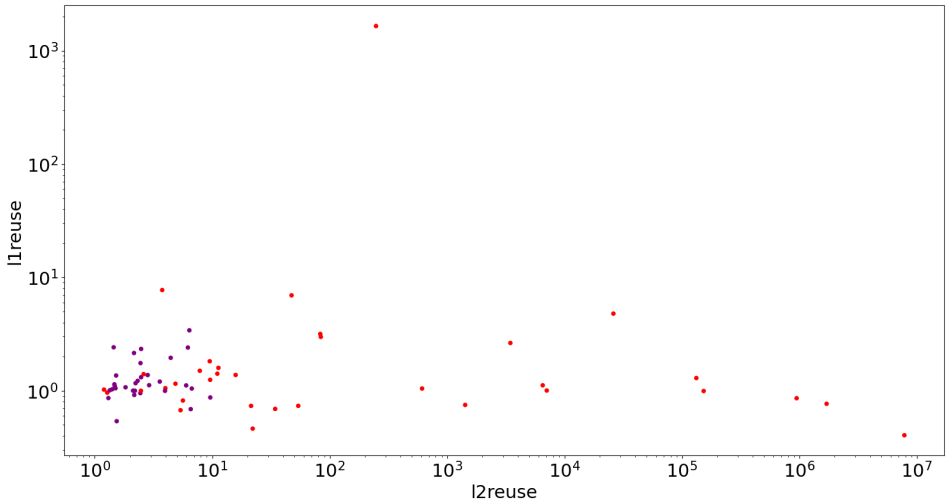

Figura 4.13: En rojo, los *benchmarks* no intensivos en memoria, si tomamos el umbral `llcm_bandwidth=200`. Gráfico en escala logarítmica.

En efecto, podemos ver como los puntos rojos y morados están más mezclados que en el primer gráfico. Cambiando de tercio, que el conjunto de trabajo quepa en la caché L3 significa que la aplicación en cuestión se beneficia al máximo de la caché, pues logra que la información necesaria esté presente en la caché cuando se la necesita (casi siempre), reduciendo al máximo los accesos a memoria. Es de esperar que particionar en este caso no produzca resultados positivos, pues la tasa de fallos caché tiene muy poco margen de mejora, pero mucho margen para empeorar. Los valores de *speedup* de los *benchmarks* de la clase confirman empíricamente esta idea intuitiva: Ninguno mejora significativamente pero si hay varios que empeoran significativamente su rendimiento, mientras la mayoría obtienen un rendimiento muy similar al caso de caché compartida (en algunos casos porque sus conjuntos de trabajo caben en cachés de nivel superior, y por tanto no dependen de la L3). Evaluamos ahora el conjunto de los *benchmarks* no intensivos en memoria (`llcm_bandwidth < 100`), cuyos valores de *speedup* se muestran en la figura 4.14.

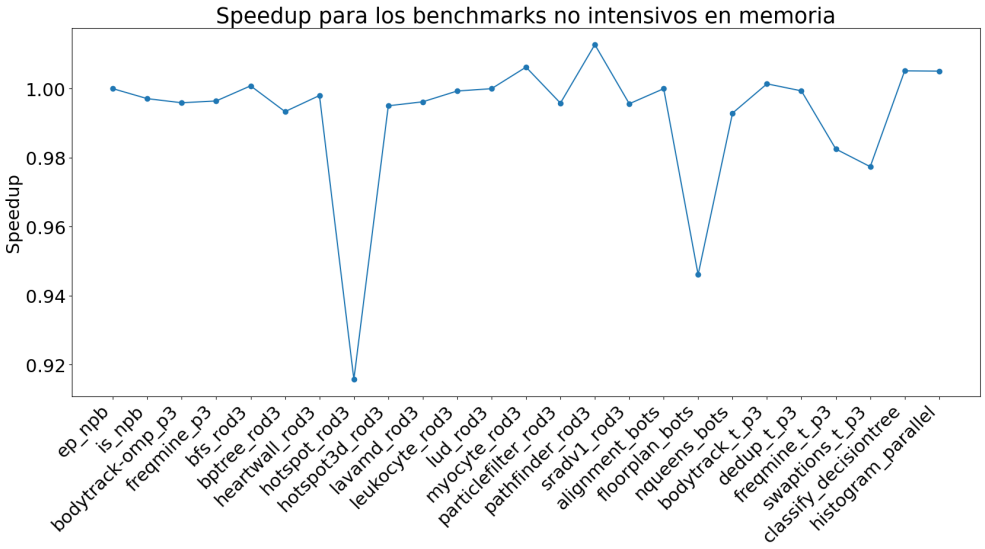

Figura 4.14: *Speedup* de los *benchmarks* no intensivos en memoria.

Si buscamos correlaciones de las métricas con el *speedup* obtenemos lo siguiente:

llcrpkc	llcmpkc	llcm_bandwidth	l1bw	l2bw	l1reuse	l2reuse
-0.274063	0.195122	0.209023	-0.742654	-0.391707	-0.153811	-0.089621

Observamos que hay una correlación inversa moderada entre `l1bw` y *speedup*. También entre `l2bw` y *speedup*. Esto significa que cuando hay pocos fallos de caché L1 y/o L2 el *speedup* tiende a ser más alto. Esto tiene sentido, pues si hay pocos fallos de caché L1/L2 es porque el conjunto de trabajo cabe en esas cachés y por tanto es de esperar que no se degrade el rendimiento al particionar la L3. Como no tenemos aplicaciones que mejoren significativamente, un *speedup* de 1 es alto en nuestro caso (porque es mayor que el *speedup* de la inmensa mayoría de aplicaciones no intensivas en memoria que mostramos en la figura 4.14). Esta idea está apoyada por las débiles correlaciones del *speedup* con `llcm_bandwidth`, `llcmpkc` y `llcrpkc` (esta última inversa).

Ahora evaluamos el conjunto de los *benchmarks* intensivos en memoria. Empezamos por buscar correlaciones de las variables con el *speedup*, sin demasiada fortuna:

llcrpkc	llcmpkc	llcm_bandwith	l1bw	l2bw	l1reuse	l2reuse
0.142564	0.135788	0.130664	0.022456	0.084537	-0.050006	0.018918

Como vemos, no hay ninguna correlación que podamos aprovechar. La figura 4.15 muestra el *speedup* para las aplicaciones intensivas en memoria.

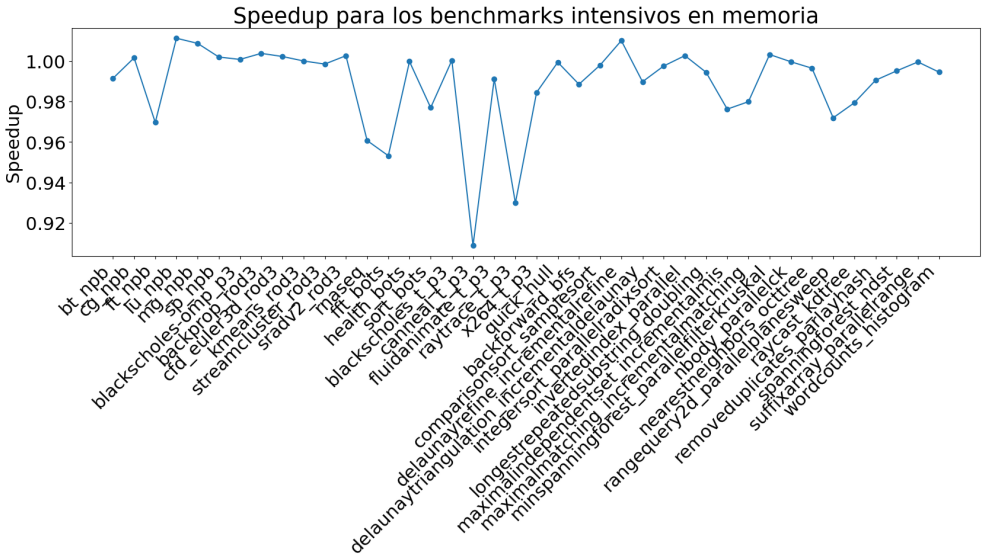

Figura 4.15: *Speedup* de los *benchmarks* intensivos en memoria.

Finalmente vamos a evaluar cómo evoluciona la frecuencia de acceso a memoria de los *benchmarks* cuando se habilita el particionado. Para esto volvemos a obtener información de contadores y calcular métricas, pero esta vez con el particionado estático activo. En la figura 4.16 se presenta el ancho de banda (`llcm_bandwidth` para cada *benchmark* (información agregada por aplicación), pintando en azul los valores correspondientes a la ejecución con caché compartida, y en naranja a la ejecución bajo particionado estático. En la figura 4.17 hacemos lo mismo con `llcmpkc`.

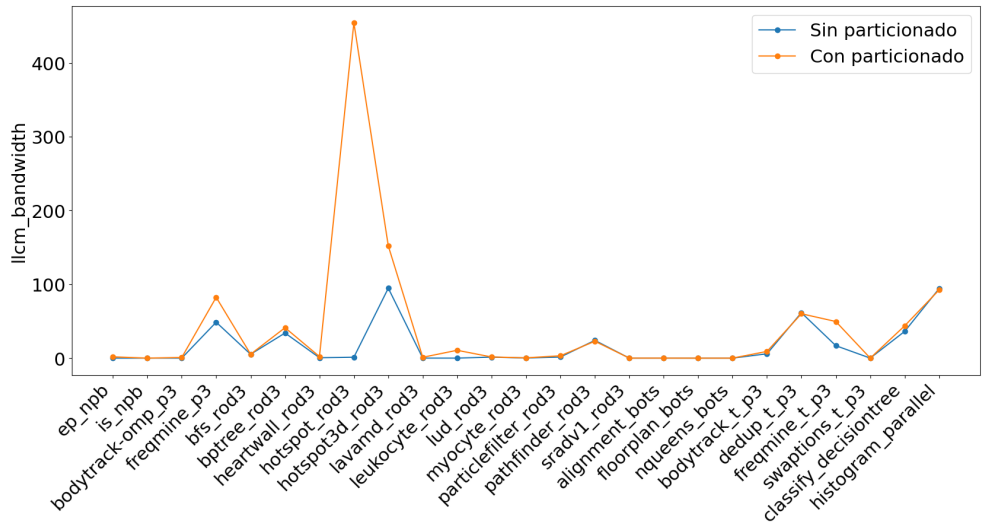

Figura 4.16: Variación del ancho de banda de la caché L3 en los *benchmarks* no intensivos en memoria.

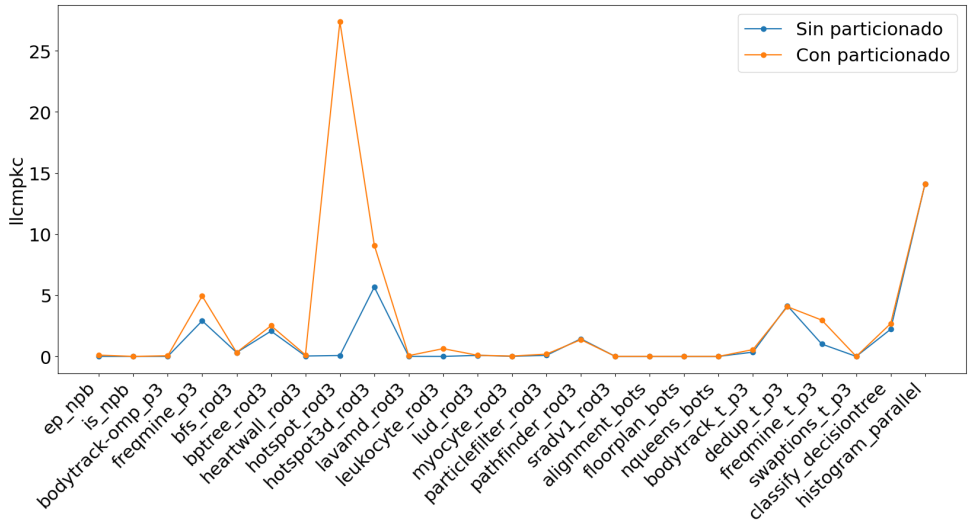

Figura 4.17: Variación de `llcmpkc` en los *benchmarks* no intensivos en memoria.

Las figuras 4.16 y 4.17 son idénticas puesto que `llcmpkc` y `llcm_bandwidth` están correlacionadas. Estás gráficas confirman nuestras hipótesis anteriores de que no conviene particionar por hilo en aplicaciones no intensivas en memoria: Ninguno de los *benchmarks* mejora en términos de `llcmpkc` y en el caso de `hotspot_rod3` es la aplicación con la mayor degradación de rendimiento tanto en *speedup* como en aumento de `llcmpkc`. Esta aplicación pasa de tener un número de fallos de caché L3 (por cada mil ciclos) prácticamente nulo a unas cifras que le llevarían a ser catalogada como intensiva en memoria. Todos los *benchmarks* que empeoran en la figura 4.16 empeoran en términos de *speedup*, aunque no parece haber una correlación demasiado fuerte. De nuevo, como ha sido la tónica general de los resultados obtenidos, la mayoría de las aplicaciones no muestran diferencias significativas entre particionar o no.

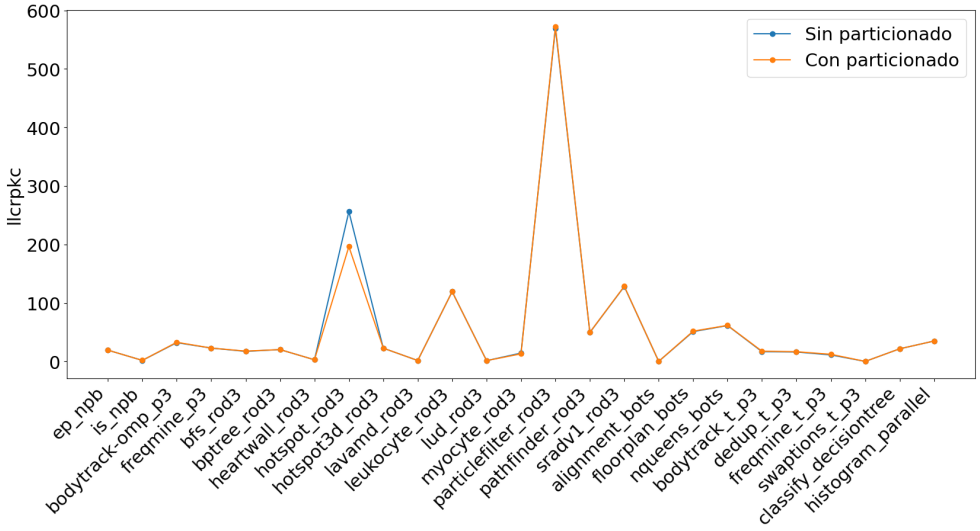

Figura 4.18: Variación de `llcrpkc` en los *benchmarks* no intensivos en memoria.

En la figura 4.18 podemos ver la variación entre particionar estáticamente o no, en términos de `llcrpkc` (*LLC Requests per Kilo Cycle*). Como es de esperar no hay diferencias salvo en el caso precisamente de `hotspot_rod3`, que muestra una reducción de `llcrpkc`. Esto se debe a que al aumentar tanto el `llcmpkc` aumenta el tiempo que se pasa cada 1000 ciclos reemplazando líneas, con lo que, cada mil ciclos se hacen menos referencias a caché L3. Hay que tener en cuenta que el *benchmark* tiene un tiempo de ejecución mucho más largo al particionar, con lo que se ejecuta muchos más ciclos.

Hacemos lo mismo para las aplicaciones intensivas en memoria, y lo presentamos en la figura 4.19 (`llcm_bandwidth`) y la figura 4.20 (`llcmpkc`).

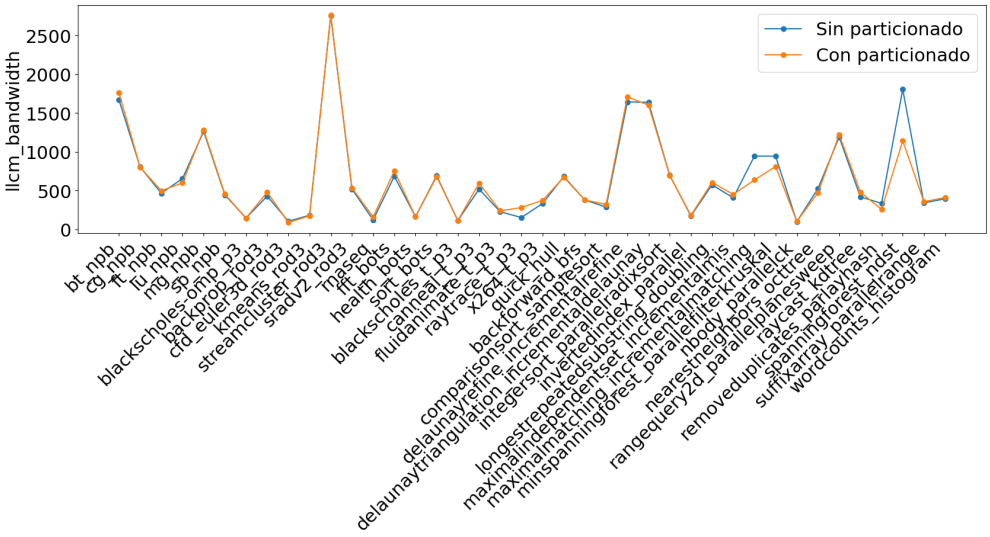

Figura 4.19: Variación del ancho de banda de la caché L3 en los *benchmarks* intensivos en memoria.

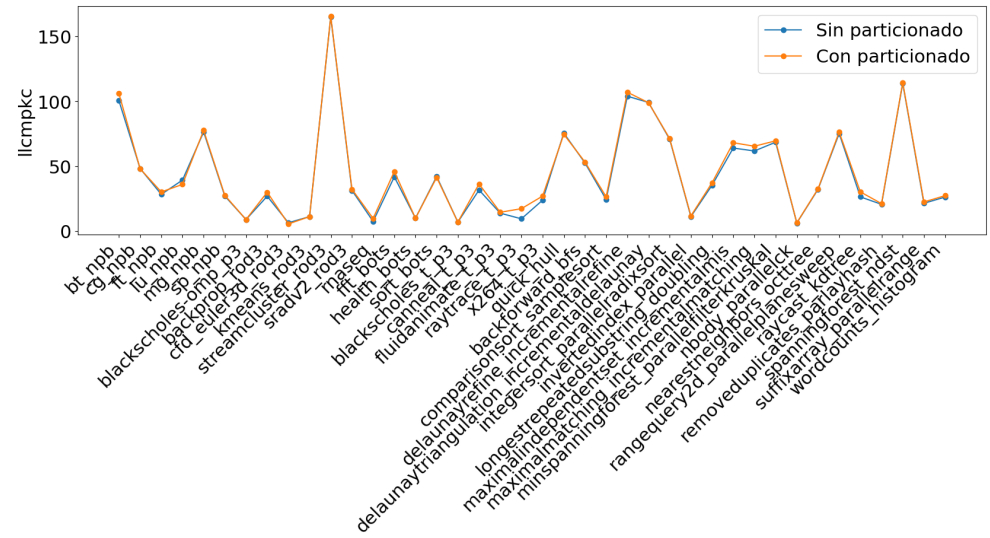

Figura 4.20: Variación de `llcmpkc` en los *benchmarks* intensivos en memoria.

En este caso las figuras 4.19 y 4.20 presentan diferencias importantes. En 4.19 podemos ver como `maximalmatching_incrementalmatching`, `minspanningforest_parallelfilterkr` y `spanningforest_ndst` reducen sustancialmente su ancho de banda, lo que no se refleja en ninguno de los casos en diferencias en términos de `llcmpkc` (todos empeoran ligeramente) en la figura 4.20. Por último, la figura 4.21 muestra la evolución de los valores de `llcrpkc`, sin que se aprecien diferencias reseñables.

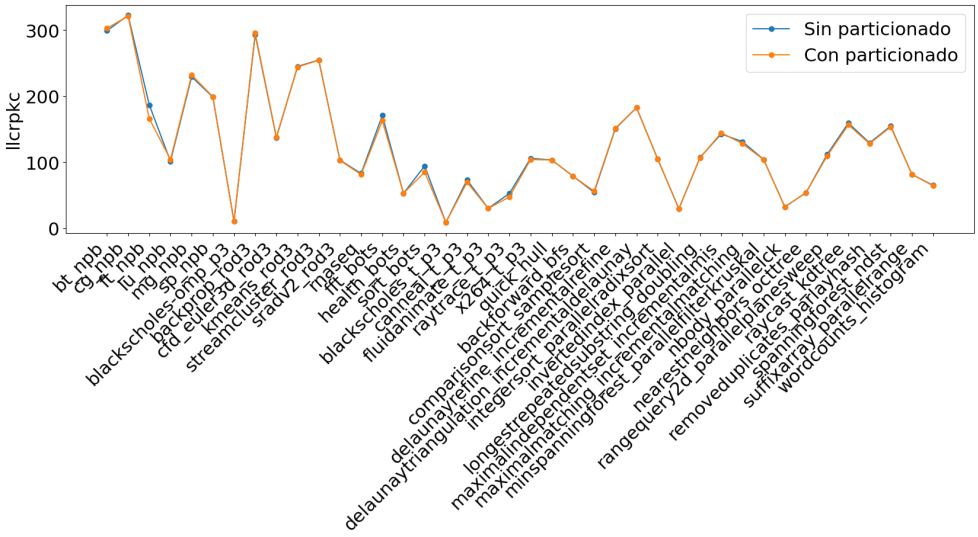

Figura 4.21: Variación de `llcrpkc` en los *benchmarks* intensivos en memoria.

4.4.3. Análisis de datos (por hilo)

Hasta ahora hemos realizado nuestro análisis utilizando como datos de las aplicaciones el agregado de los datos de sus hilos. Ahora vamos a proceder a aplicar la clasificación que hemos diseñado a los hilos individuales de cada *benchmark*.

En la figura 4.22 aparecen los *benchmarks* que se ejecutan con 8 hilos en paralelo (que son prácticamente todos, pues como comentamos anteriormente pasamos a las aplicaciones un parámetro con este fin), y muestra de qué tipo (intensivo en memoria o no intensivo en memoria) es cada uno. El umbral utilizado para clasificar los hilos es `llcm_bandwidth=100`, que es el mismo que utilizamos a la hora de clasificar por aplicación. Observamos que casi todas las aplicaciones presentan un patrón de acceso homogéneo: Todos sus hilos caen en la misma clase. Esto significa que con los datos de los que disponemos no vamos a poder realizar ninguna clasificación basándonos en el comportamiento individual de cada hilo.

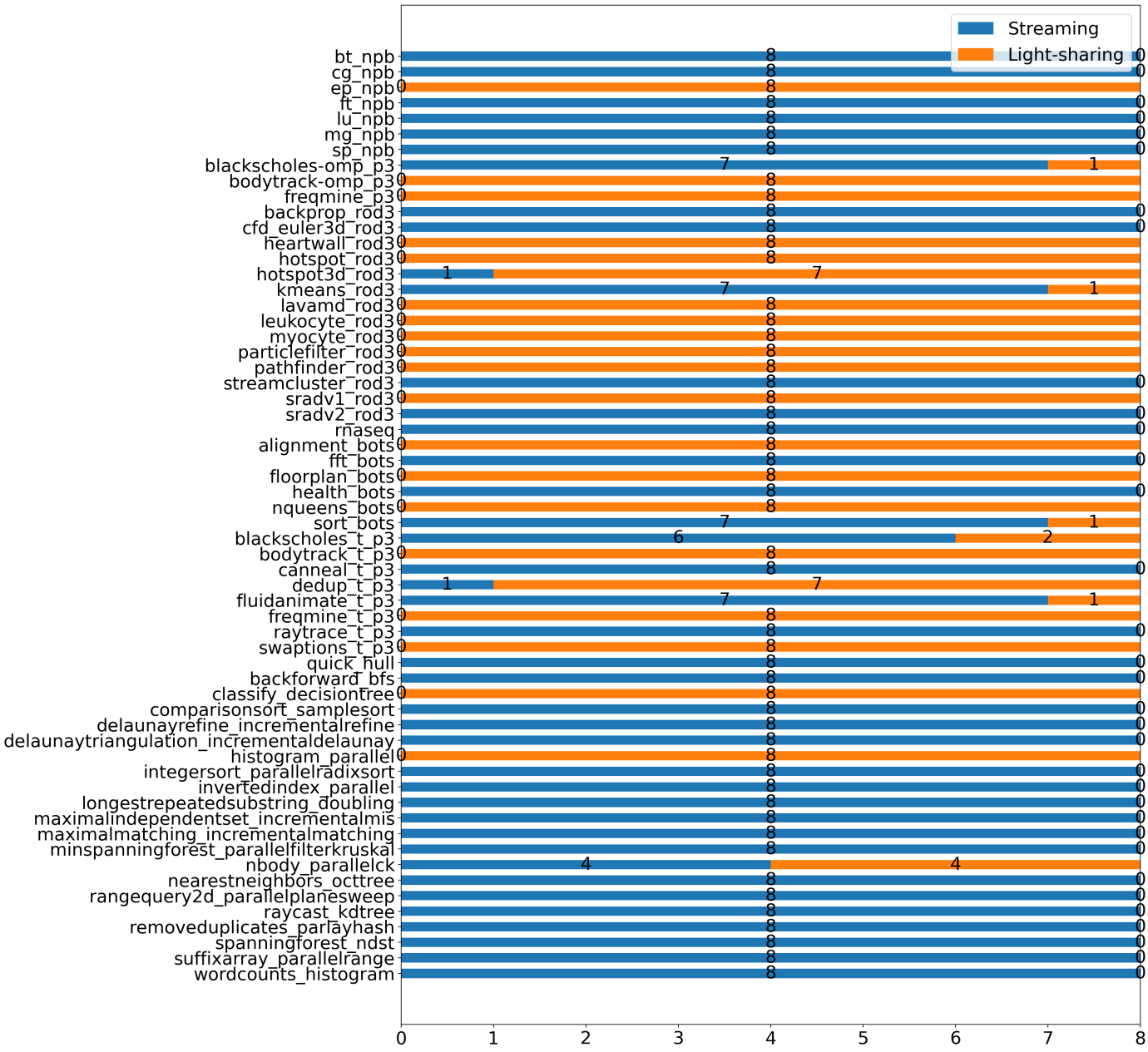

Figura 4.22: Para cada *benchmark* se muestra la clase (intensivo o no intensivo en memoria) de cada uno de sus hilos. Un hilo es considerado como no intensivo en memoria si `llcm_bandwidth < 100`.

Pasamos a enumerar y comentar brevemente las aplicaciones que han mostrado algunas diferencias (en las figuras 4.22, en 4.16 o en 4.19), y también aquellas que

empeoren significativamente en *speedup*:

- `raytrace_t_p3`: Todos sus hilos son intensivos en memoria, pero mirando los datos de `llcrpkc`, que se presentan por hilo en la figura 4.23, podemos ver que el hilo `master` (`thread id = 0`) hace muchos más accesos a la LLC que el resto. Además su `l2reuse` es mucho más alto, como se ve en la figura 4.24.

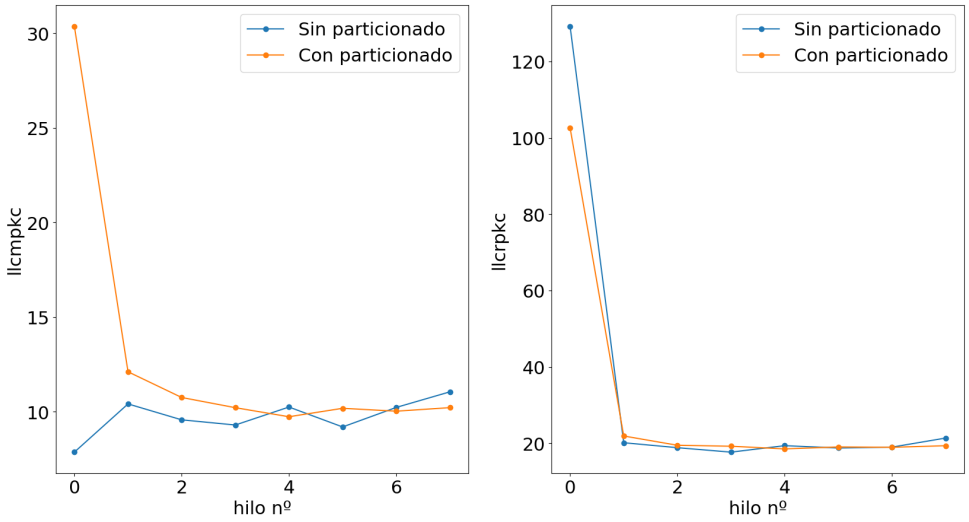

Figura 4.23: Variación de `llcmpkc` y `llcrpkc` por hilo.

En este caso particionar provoca que el hilo principal, que se estaba beneficiando del acceso a toda la caché, empeore de manera muy notoria (como vemos en el valor de `llcmpkc`, también en la figura 4.23) mientras el resto de hilos no se benefician del particionado. Los valores de `llcrpkc` para el hilo principal son más bajos porque estamos pasando más tiempo de media cada mil ciclos reemplazando líneas en la caché L3.

Además, la métrica `l2reuse` se desploma en el hilo `master`, lo que de nuevo confirma que el rendimiento en el acceso a memoria se deteriora considerablemente. Por completitud añadimos la evolución de `llcm_bandwidth` (en la figura 4.24), aunque es idéntico al de `llcmpkc`. Conjeturamos que particionar (estáticamente) cuando los hilos presentan patrones distintos de acceso a memoria provoca pérdidas en el rendimiento. Cabe preguntarse si particionando de otra forma (por ejemplo, dando la mayor parte de la caché al hilo principal) puede conseguirse alguna mejoría.

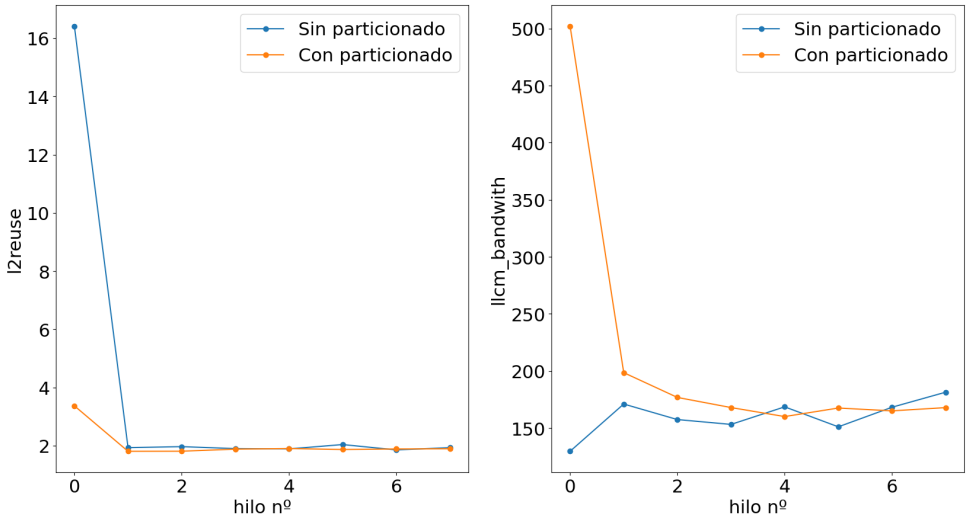

Figura 4.24: Variación de `l2reuse` y `llcm_bandwidth` por hilo.

- `canneal_t_p3`: Presenta la misma situación que `raytrace_t_p3`, aunque la diferencia entre el hilo principal y el resto de hilos no es tan grande. Además, en este caso todos los hilos empeoran, si bien sólo el principal lo hace de forma notable (figura 4.25).

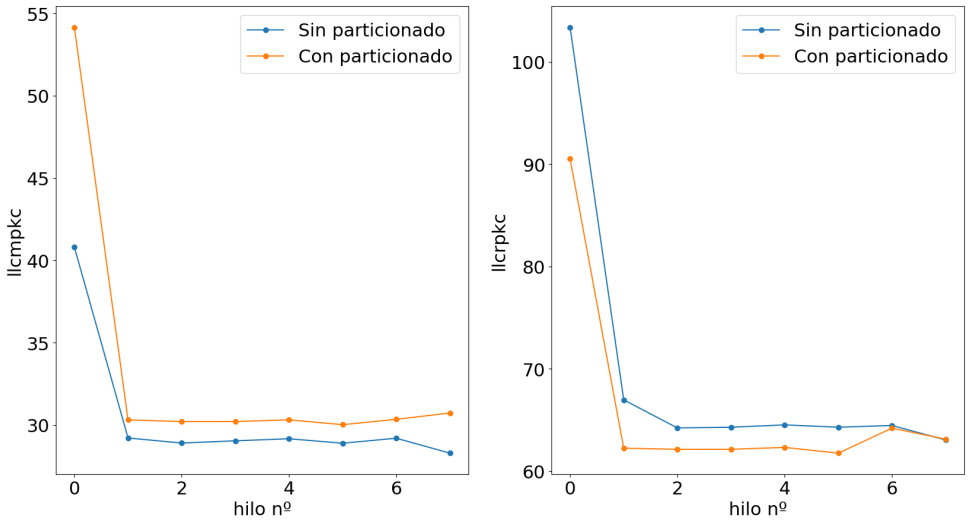

Figura 4.25: Variación de `llcmpkc` y `llcrpkc` por hilo (`canneal_t_p3`).

El valor de `l2reuse` del hilo principal no cae en exceso, como vemos en la figura 4.26, pues ya partíamos de un valor relativamente bajo. Como curiosidad comentar que el valor de `stalls_l2_miss` (también en la figura 4.26) del hilo principal es muy superior al del resto de hilos (que es aproximadamente 0),

siendo esta una de las pocas aplicaciones que muestra algún valor reseñable en esta métrica.

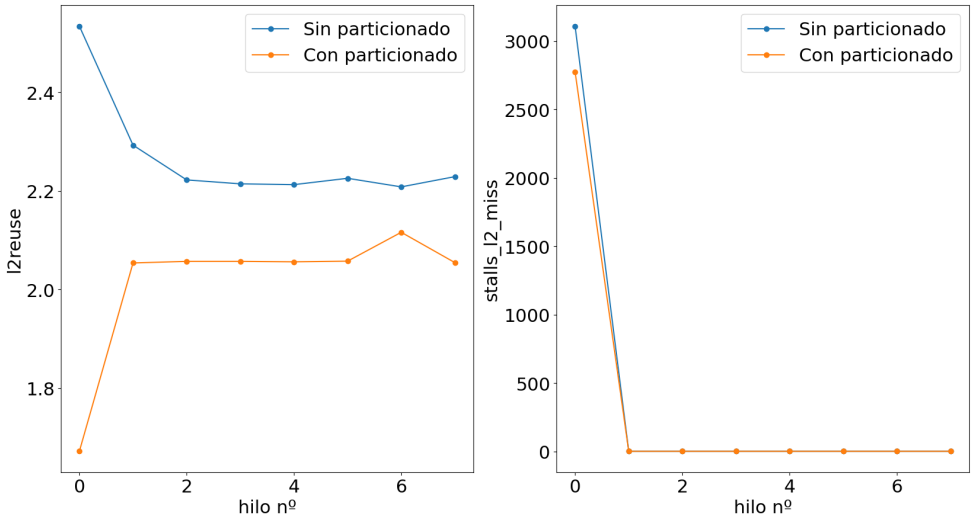

Figura 4.26: Variación de `l2reuse` y `stalls_l2_miss` por hilo (`canneal_t_p3`).

- `hotspot_rod3`: Como ya hemos comentado antes esta aplicación pasa de tener apenas fallos de caché (`llcmpkc` aproximadamente 0) a ser una aplicación con un ancho de banda elevadísimo (figura 4.27). En este caso el comportamiento de los hilos es más homogéneo.

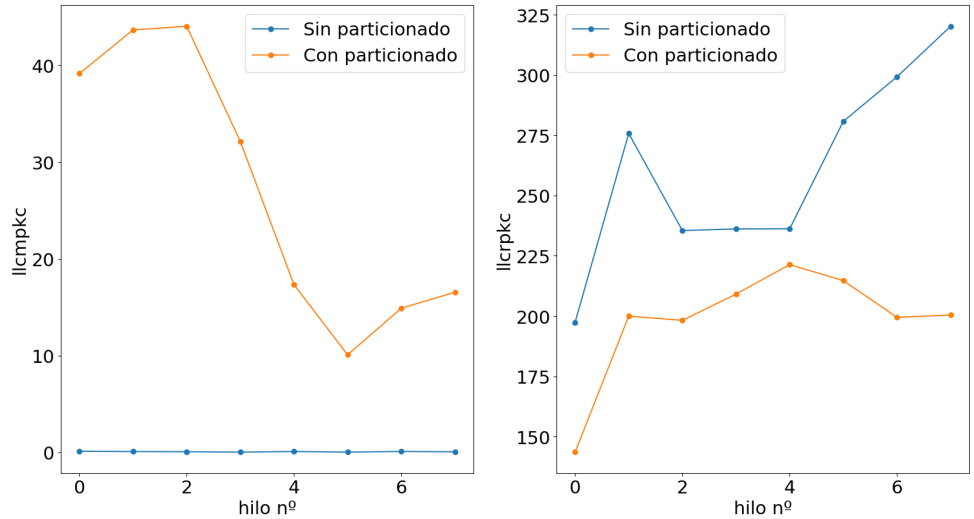

Figura 4.27: Variación de `llcmpkc` y `llcrpkc` por hilo (`hotspot_rod3`).

En la figura 4.28 vemos como los valores de `l2reuse` varían mucho de unos hilos a otros (sin particionado) pero en cualquier caso esto confirma que al particionar pasa de ser una aplicación que hace un reuso alto de la información

de la caché L3 a comportarse como una aplicación *streaming*. Parece claro que el conjunto de trabajo cabía en la caché L3 (`llcrpkc` alto y `llcmpkc` bajo) y que al particionarla los hilos han empezado a "pisarse" unos a otros. Lo cual no deja de ser paradójico porque el objetivo del particionado es precisamente aumentar el grado de aislamiento entre hilos. Hay que tener presente que el particionado es sólo para el reemplazo de líneas, no para el acceso a la información de la caché. Esto puede provocar que una línea de la caché muy usada sea reemplazada habiendo en la caché otras líneas mucho menos usadas (basta con que sea la menos usada de su partición).

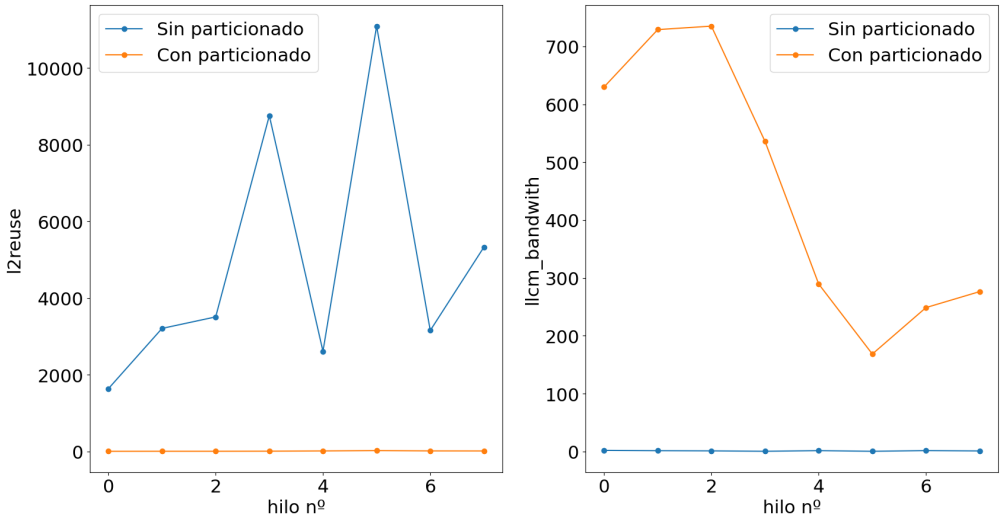

Figura 4.28: Variación de `l2reuse` y `llcm_bandwidth` por hilo (`hotspot_rod3`).

- `floorplan_bots`: Aquí la diferencia en ancho de banda de la caché L3 es mínima, como vemos en la figura 4.29. Recordamos que este era uno de los *benchmarks* que empeoraban significativamente (*speedup* $\approx 0{,}946$).

  En la figura 4.30 se muestra la variación de `l2reuse` y `llcmpkc` entre particionar o no, por hilo, para `floorplan_bots`. Nótese que si bien este es uno de esos *benchmarks light-sharing* en los que hemos planteado que particionar no va a ser una buena estrategia porque tienen una tasa de fallos de caché L3 casi óptima (en el caso de `floorplan_bots` su `llcmpkc` medio por hilo es aproximadamente 0) no es esperable una degradación tan grande del rendimiento, pues aunque el aumento de los fallos de L3 es un factor de 2, o incluso de 3, en términos absolutos siguen siendo despreciables. Este factor provoca una variación enorme de la métrica `l2reuse` (porque `l2reuse = fallos L2 / fallos L3`) pero que no implica que tenga que darse una degradación de rendimiento.

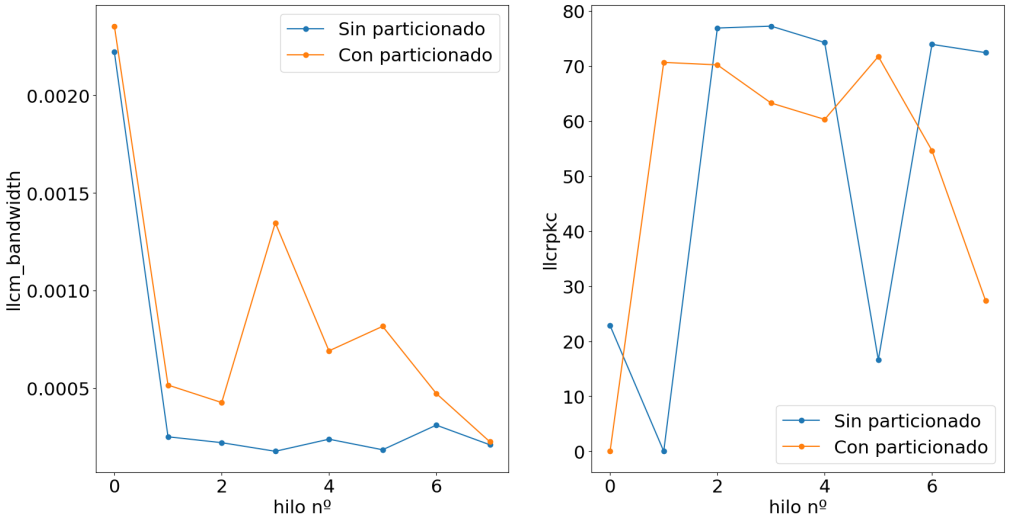

Figura 4.29: Variación de `llcm_bandwidth` y `llcrpkc` por hilo (`floorplan_bots`).

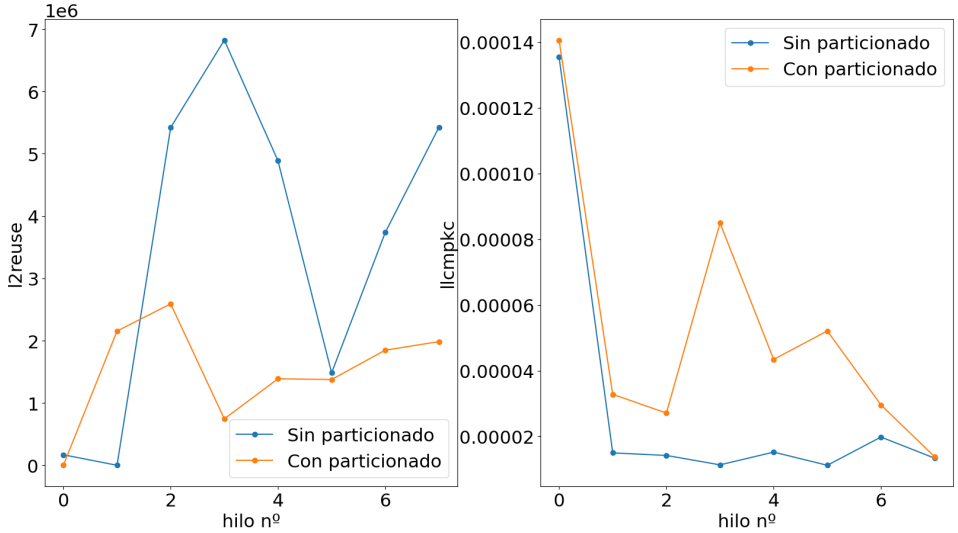

Figura 4.30: Variación de `l2reuse` y `llcmpkc` por hilo (`floorplan_bots`).

Una vez que hemos terminado de analizar los *benchmarks* que empeoran significativamente vamos a realizar un análisis similar con aquellos que presentan tanto hilos intensivos en memoria como no intensivos en memoria en la figura 4.22.

- `blackscholes-omp_p3`: El hilo principal es clasificado como no intensivo en memoria, como vemos en la figura 4.31 (que muestra la variación de `llcm_bandwidth` y `llcrpkc` por hilo). Tiene un patrón de acceso distinto al resto de hilos, aunque no muestra diferencias en ninguna de las métricas entre particionar o no.

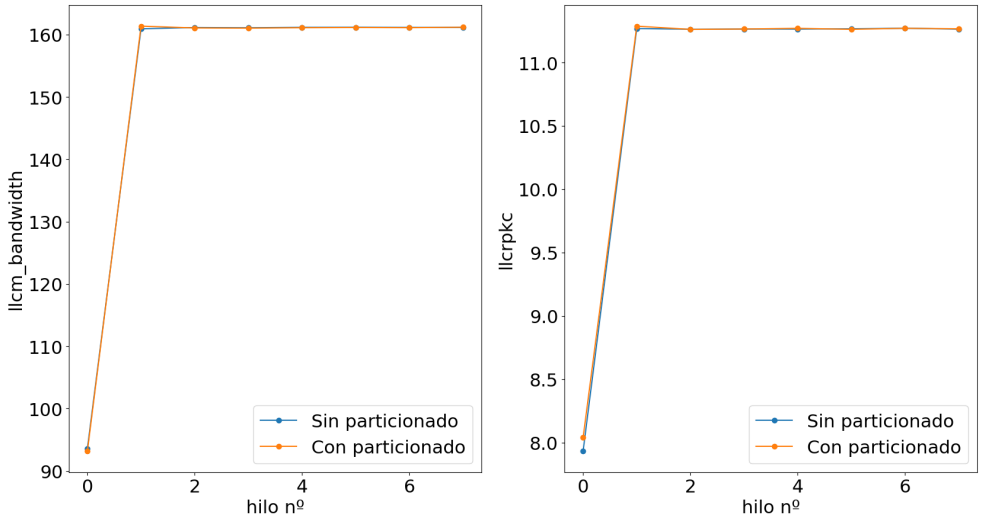

Figura 4.31: Variación de `llcm_bandwidth` y `llcrpkc` por hilo (`blackscholes-omp_p3`).

- `hotspot3d_rod3`: Las figuras 4.32 y 4.33 muestran la variación de algunas métricas al particionar. En ellas vemos que el hilo principal es intensivo en memoria mientras el resto son no intensivos. Este *benchmark* tiene un *speedup* $\approx 0{,}995$. En este caso el hilo principal pasa a exhibir un comportamiento más intensivo en memoria (`llcmpkc` con particionado es más del doble que sin particionado) mientras los demás hilos muestran una ligera mejora, en forma de un `llcmpkc` más bajo y `l2reuse` ligeramente más alto. Cabe recordar que `hotspot3d_rod3` es uno de los pocos *benchmarks* que mostraba peores datos de `llcmpkc` al particionar en la figura 4.16, lo que como vemos es debido a que la pequeña mejora de los hilos no intensivos en memoria al particionar no compensa el notable empeoramiento del hilo principal. Esto puede estar debido a que, como ya hemos comentado anteriormente, los hilos no intensivos de esta aplicación tienen muy pocos fallos de caché, con lo que en cualquier caso el beneficio máximo que se puede obtener es muy limitado. Por otro lado, el hilo principal es mucho más intensivo en memoria y es razonable que empeore al recibir una porción muy pequeña de la caché L3 (apenas 3 vías, de un total de 20).

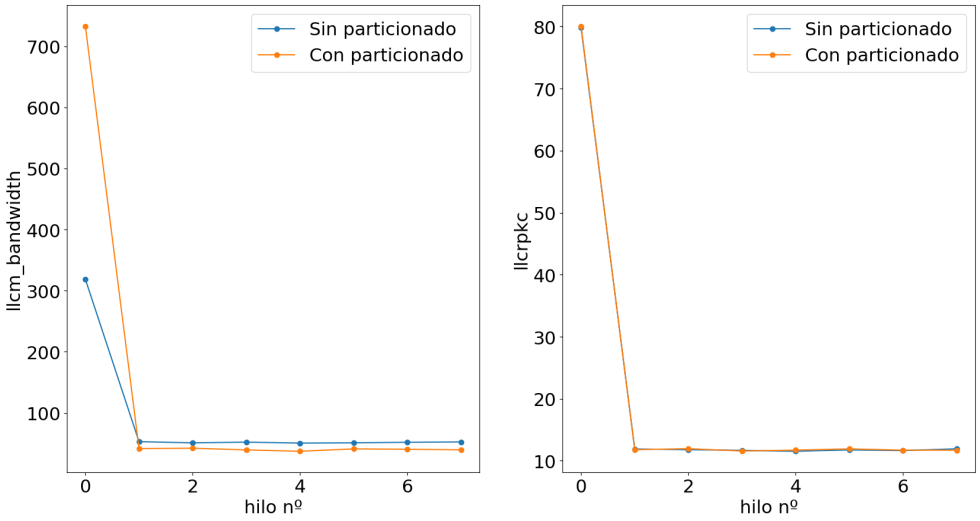

Figura 4.32: Variación de `llcm_bandwidth` y `llcrpkc` por hilo (`hotspot3d_rod3`).

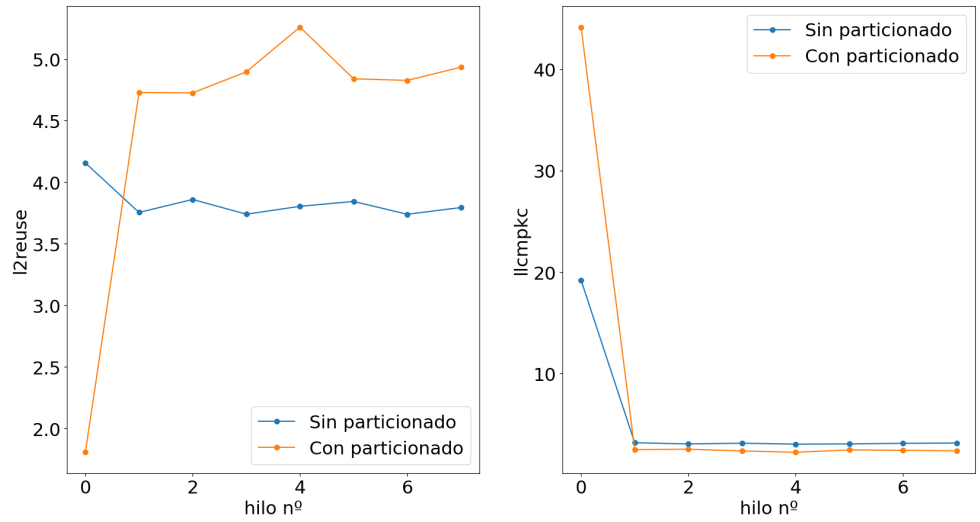

Figura 4.33: Variación de `l2reuse` y `llcmpkc` por hilo (`hotspot3d_rod3`).

- `kmeans_rod3`: En la figura 4.34, que muestra la variación de `llcm_bandwidth` y `llcrpkc`, observamos que el hilo principal tiene un comportamiento menos intensivo en memoria que el resto. De hecho, el hilo principal se clasifica como no intensivo, aunque está muy cerca del límite. El resto de hilos muestran un comportamiento más intensivo en memoria, pero muy similar entre ellos. No se aprecia ninguna diferencia (en términos de métricas) entre particionar estáticamente o no particionar y de hecho *speedup* $\approx 1{,}0$.

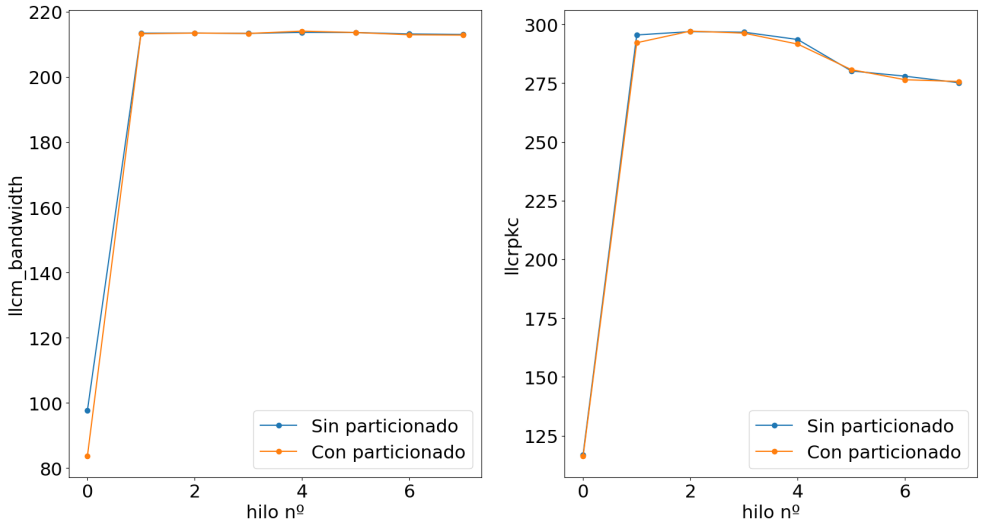

Figura 4.34: Variación de `llcm_bandwidth` y `llcrpkc` por hilo (`kmeans_rod3`).

- `sort_bots`: Aquí encontramos dos hilos distinguidos. Por un lado el hilo principal (hilo número 0), que es más intensivo en memoria que el resto (mayor `llcrpkc` y `llcm_bandwidth`) y por otro un hilo que es no intensivo (hilo número 4, sin particionado), y que apenas accede a memoria (`llcrpkc` $\approx 0{,}01$). El particionado provoca pocos cambios. Los hilos que no son ni el 0 ni el 4 mantienen un rendimiento similar, mientras el hilo principal aumenta sus fallos de caché (referencias se mantienen constantes) y como consecuencia su ancho de banda. El hilo no intensivo (que pasa con el particionado a ser el hilo número 5) disminuye por un factor de 10 su `llcmpkc` y ancho de banda. Como consecuencia su `l2reuse` se dispara. Este *benchmark* realiza una ordenación de una permutación aleatoria de n números de 32 bits, resolviendo el problema recursivamente (dividiendo el *array* en dos partes en cada paso) y de forma paralela [31]. Por tanto, parece claro que al procesar cada hilo una parte de los datos con la que ningún otro hilo está trabajando el particionado no produce una degradación del rendimiento en esos hilos, siempre y cuando la partición que le toque a cada hilo sea suficiente para contener el conjunto de trabajo de ese hilo. En el caso del hilo número 5 tener su propia partición le permite evitar la polución de la caché que provocan el resto de hilos (que son intensivos en memoria). El hilo principal accede mucho más a memoria que los demás hilos (su `llcrpkc` es el doble que el resto) así que es probable que necesite una partición un poco más grande que las 3 vías que le asigna la estrategia de particionado estático. En cualquier caso su degradación de rendimiento es pequeña, como se puede juzgar en las figuras 4.35 y 4.36. Nótese que el hilo no intensivo cambia de `tid` (*thread ID*) en cada ejecución.

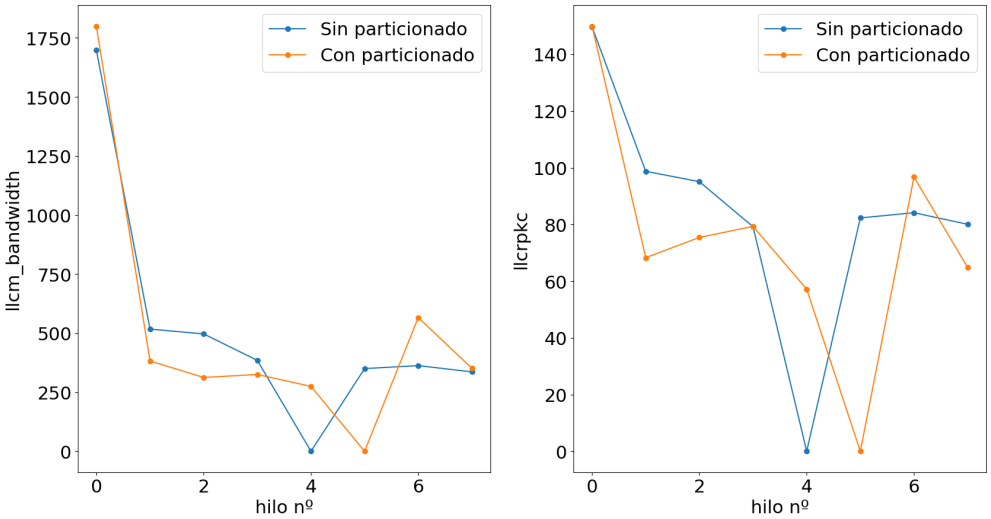

Figura 4.35: Variación de `llcm_bandwidth` y `llcrpkc` por hilo (`sort_bots`).

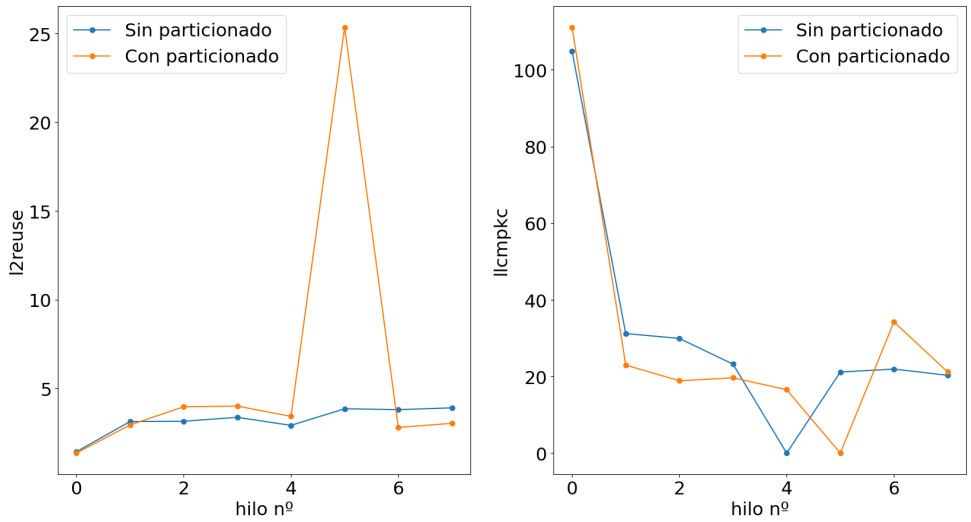

Figura 4.36: Variación de `l2reuse` y `llcmpkc` por hilo (`sort_bots`).

Para los próximo cuatro *benchmarks* hemos omitido las figuras para evitar alargar todavía más esta memoria. Se trata de figuras muy similares a otras mostradas anteriormente (pues estos *benchmarks* no muestran diferencias entre particionar o no), por lo que no aportan demasiado, más allá de constatar este hecho.

- `blackscholes_t_p3`: En este caso tenemos 6 hilos intensivos en memoria y prácticamente idénticos en sus métricas. Los otros 2 hilos quedan clasificados como no intensivos en memoria, aunque uno de ellos está cerca del límite `llcm_bandwidth=100`. Todos los hilos de esta aplicación exhiben en cualquier

caso un comportamiento `streaming`, pues el porcentaje de fallos de caché (`llcmpkc / llcrpkc`) es muy alto (en torno al 80 % en cada hilo) lo que queda reflejado en una bajísima métrica de reuso (todos los hilos se mueven entre 1,2 y 1,45). No se observan cambios en el rendimiento (*speedup* $\approx$ 1,0) ni en las métricas con el particionado.

- `dedup_t_p3`: En este vemos como un hilo tiene un comportamiento claramente mucho más intensivo que el resto, que son no intensivos en memoria. Esto se debe a la naturaleza asimétrica de este *benchmark*, que se encarga de comprimir un flujo de datos de forma concurrente con un hilo encargándose de separar el flujo en trozos (secuencialmente) que el resto de hilos pueden comprimir y que finalmente son unidos de nuevo por el mismo hilo inicial. No se observan cambios en el rendimiento (*speedup* $\approx$ 1,0) ni en las métricas con el particionado.

- `fluidanimate_t_p3`: Aquí tenemos un hilo (el hilo padre) que es no intensivo en memoria, mientras el resto de hilos sí lo son. Aquí tampoco se observan cambios en el rendimiento (*speedup* $\approx$ 0,99) ni en las métricas al particionar.

- `nbody_parallelck`: La mitad de sus hilos son no intensivos en memoria y la otra mitad intensivos en memoria, pero con un simple vistazo a sus métricas podemos comprobar que se trata de un *benchmark* cuyos hilos se mueven en la frontera de `llcm_bandwith=100` que hemos establecido, pero su patrón de acceso a memoria es similar.

Para acabar, vamos a analizar las aplicaciones que no hayamos comentado todavía y que muestren diferencias en las figuras 4.16 y 4.19, al ejecutarse con particionado estático.

- `freqmine_p3`: Se trata de una aplicación no intensiva en memoria que experimenta pérdidas de rendimiento al particionar, como constata el aumento de ancho de banda de la caché L3 y la reducción de la métrica de reuso `l2reuse` en la figura 4.37.

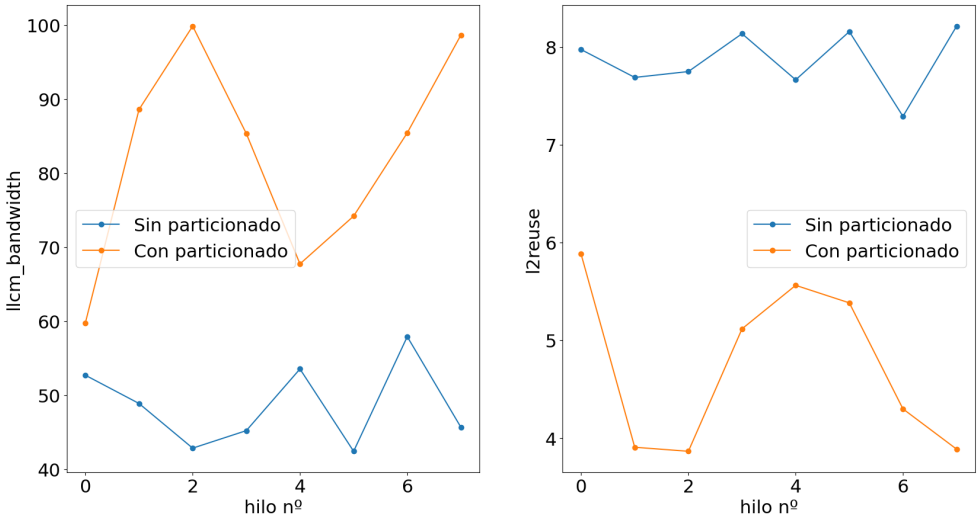

Figura 4.37: Variación de `llcm_bandwidth` y `l2reuse` por hilo (`freqmine_p3`).

- `leukocyte_rod3`: De la figura 4.38 se concluye que `leukocyte_rod3` es otra aplicación no intensiva en memoria en la que todos sus hilos tienen valores similares para las métricas. Es decir, el acceso a caché y memoria es homogéneo. Se trata de una de esas aplicaciones (como `hotspot_rod3`) con una tasa de fallos en el acceso a caché óptima (`llcm_bandwidth` y fallos de caché L3 son aproximadamente 0), pero que al particionar muestran una degradación de rendimiento importante.

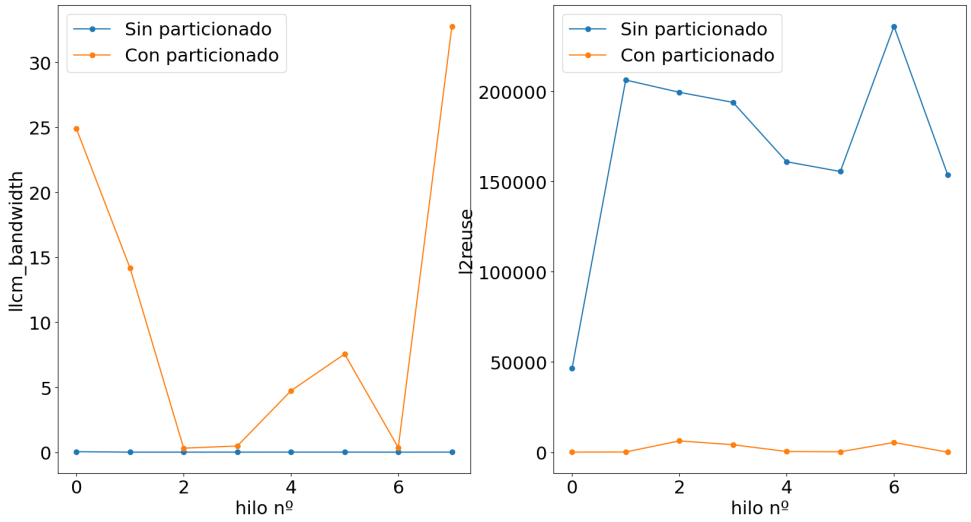

Figura 4.38: Variación de `llcm_bandwidth` y `l2reuse` por hilo (`leukocyte_rod3`).

- `classify_decisiontree`: Otra aplicación no intensiva que, como vemos en la figura 4.39, sufre una pequeña degradación en sus métricas al particionar, pero

sin llegar a afectar al rendimiento (*speedup* ≈ 1).

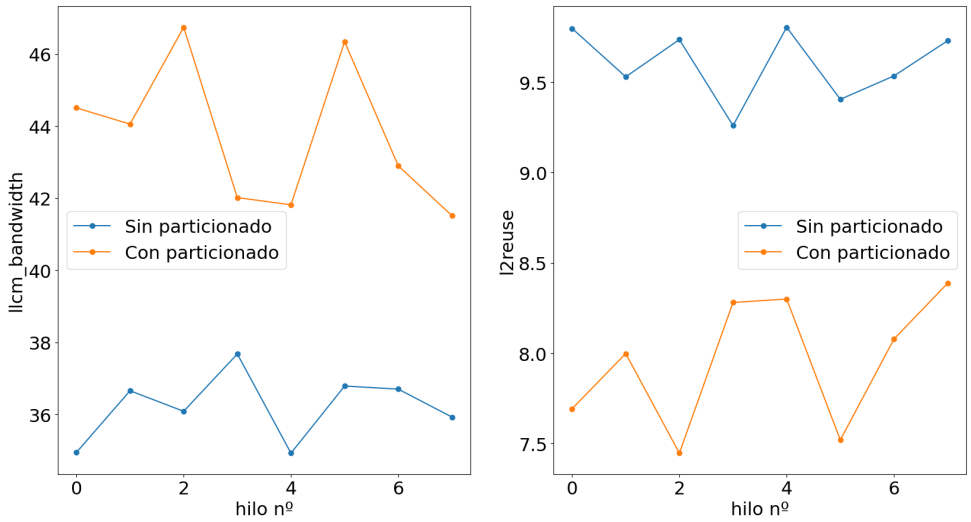

Figura 4.39: Variación de `llcm_bandwidth` y `l2reuse` por hilo (`classify_decisiontree`).

- `bt_npb`: De la figura 4.40 se concluye que esta es una aplicación muy intensiva en memoria, cuyos hilos tienen todos el mismo patrón de acceso a caché y memoria. El *speedup* es aproximadamente 1,0 y los valores de las métricas casi no cambian al aplicar el particionado. En ambos casos parece tener un comportamiento bastante `streaming`, pues casi no se aprovecha la caché (valor bajo de `l2reuse`).

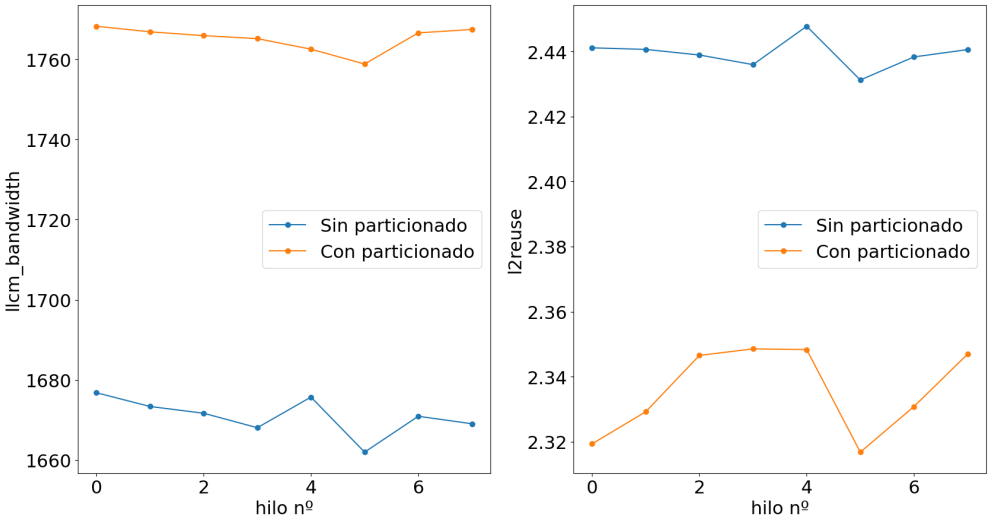

Figura 4.40: Variación de `llcm_bandwidth` y `l2reuse` por hilo (`bt_npb`).

- `lu_npb`: Echando un vistazo a 4.41 nos queda claro que estamos en una situación muy similar a la aplicación anterior, pero en este caso se produce una pequeña mejora en las métricas que no se traduce en un incremento significativo del rendimiento (*speedup* $\approx$ 1,01).

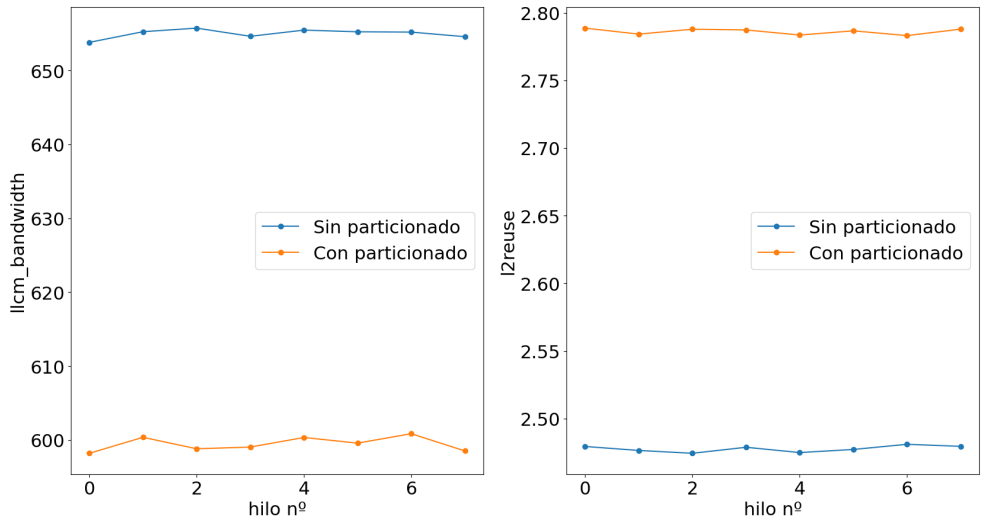

Figura 4.41: Variación de `llcm_bandwidth` y `l2reuse` por hilo (`lu_npb`).

- `backprop_rod3`: En la figura 4.42 vemos que, si bien todos los hilos de esta aplicación son intensivos en memoria, hay uno (el hilo número 5) que lo es mucho más que el resto, pues tiene casi 20 veces más fallos de caché que el resto, mientras las referencias a caché L3 (`llcrpkc`) de todos los hilos son similares (salvo el hilo principal, que hace la quinta parte de referencias a memoria que el resto de hilos).

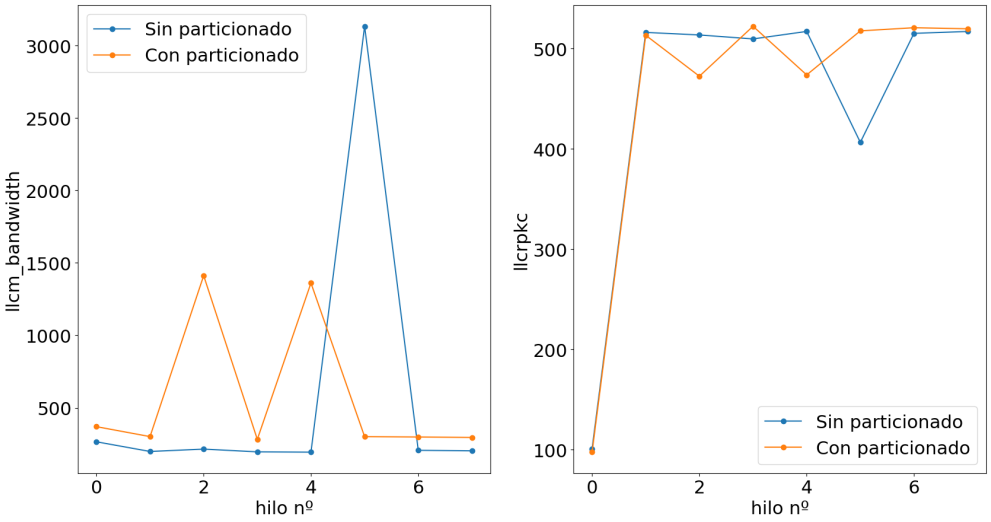

Figura 4.42: Variación de `llcm_bandwidth` y `llcrpkc` por hilo (`backprop_rod3`).

Los valores de `l2reuse` son muchísimo más bajos en este hilo "singular" que en el resto (salvo, de nuevo, el hilo principal). Particionar provoca que no haya ningún hilo con un rendimiento tan malo como el que tenía 5 antes de particionar, aunque hay 2 hilos (el 2 y el 4) que tienen rendimientos notablemente peores que el resto. Por otro lado, todos los demás hilos degradan su rendimiento. Esto de nuevo parece indicarnos que particionar utilizando *clusters* en lugar de particiones disjuntas puede ser prometedor, ya que parece que al aislar los hilos evitamos que ninguno tenga un rendimiento tan malo como 5 sin particionado, pero probablemente al ser las particiones demasiado pequeñas esto acaba degradando el rendimiento de los hilos que hacían un uso relativamente eficiente de la caché cuando esta estaba compartida. El *speedup* de este *benchmark* es 1,0.

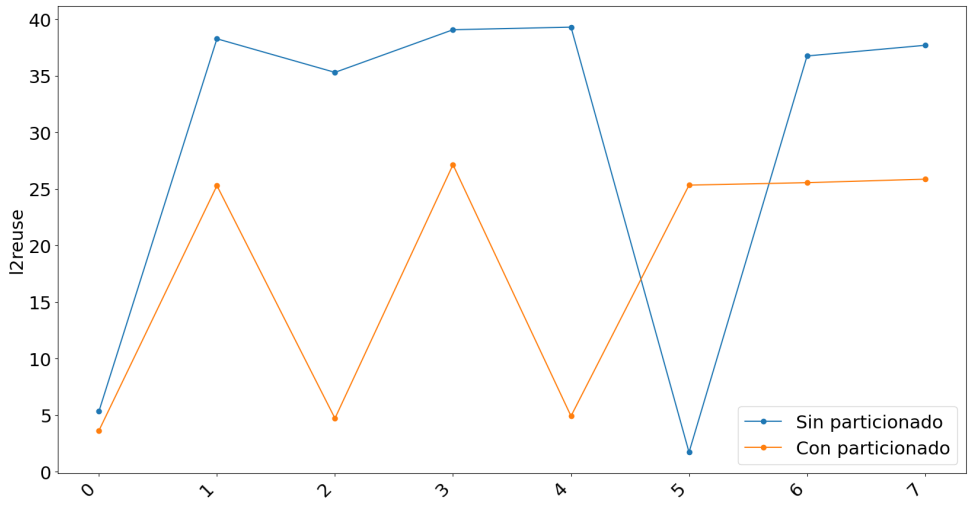

Figura 4.43: Variación de `l2reuse` por hilo (`backprop_rod3`).

- `comparisonsort_samplesort`: En la figura 4.44 se aprecia que los ocho hilos presentan un comportamiento homogéneo respecto al acceso a la memoria y la caché (en términos de las métricas medidas). Son hilos intensivos en memoria que producen ligeramente más fallos de caché bajo particionado estático, sin que llegue a afectar al *speedup* (1,0).

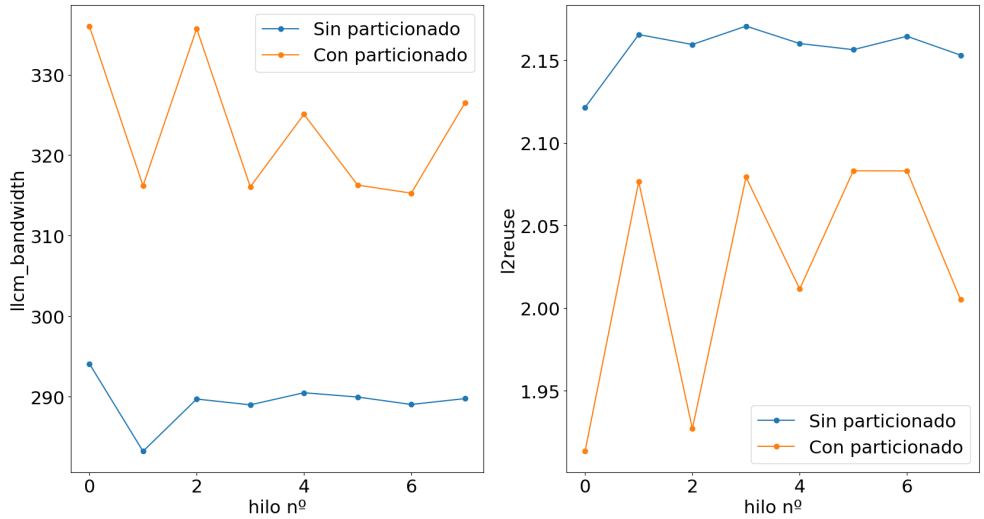

Figura 4.44: Variación de `llcm_bandwidth` y `l2reuse` por hilo (`comparisonsort_samplesort`).

- `spanningforest_ndst`: Este es el único *benchmark* junto a `maximalmatching_incrementalmatching` y `minspanningforest_parallelfilterkruskal` en el que el `llcm_bandwidth` mejora, a pesar de que el resto de resultados de las métricas son idénticas (de forma casi milimétrica), como podemos ver en las figuras 4.45 y 4.46.

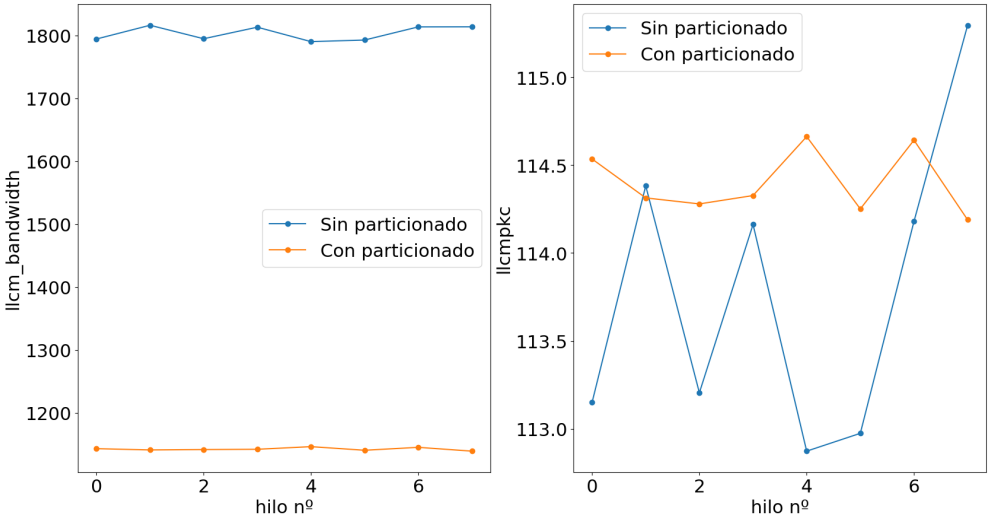

Figura 4.45: Variación de `llcm_bandwidth` y `llcmpkc` por hilo (`spanningforest_ndst`).

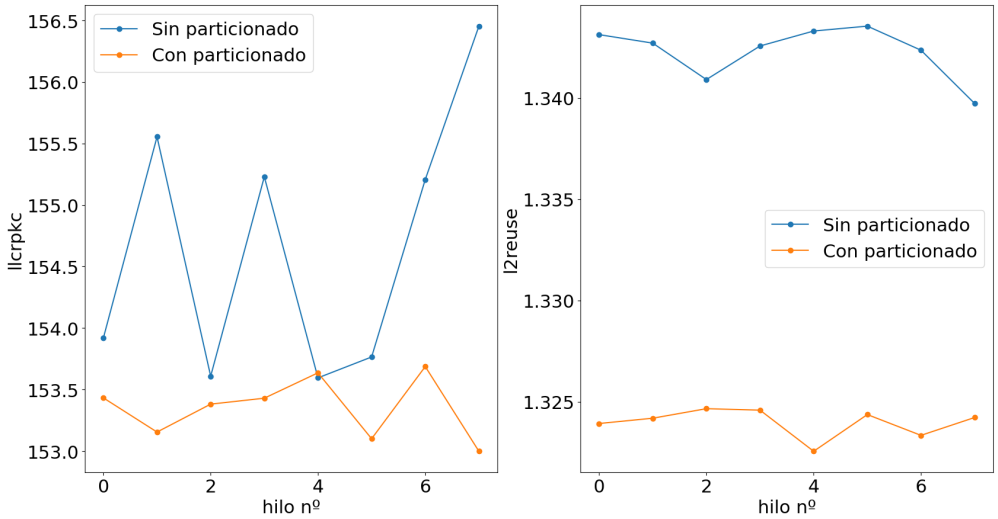

Figura 4.46: Variación de `llcrpkc` y `l2reuse` por hilo (`spanningforest_ndst`).

4.4.4. Experimentos con contenedores

Finalmente hemos realizado algunos experimentos utilizando contenedores en Broadwell-EP. Como hemos comentado anteriormente, en general, los hilos de los *benchmarks* HPC realizan todos un tipo de procesamiento similar, lo que provoca que tengan todos un patrón de acceso a caché similar, aunque en muchos casos hay un hilo `master` que carga todos los datos a memoria (y por tanto tiene un comportamiento ligeramente distinto). Por ello, hemos tratado de evaluar aplicaciones cuyos hilos presenten un comportamiento más heterogéneo, como las de la *suite* TP-Parsec [32]. El objetivo ahora es evaluar algunas aplicaciones multihilo que realicen procesamientos más independientes en cada hilo, por lo que es interesante a priori evaluar el efecto del particionado en esta clase de aplicaciones. No obstante, no vamos a presentar aquí una gran cantidad de aplicaciones evaluadas.

Hemos utilizado *benchmarks* de la *suite* Cloudsuite, que son cargas típicas de Cloud [33]. Además, hemos utilizado PBBCache, un simulador de Caché escrito en Python [35]. Este se trata de una aplicación multiproceso, no multihilo, pero que al ejecutarse dentro de un contenedor es visto por el kernel como una aplicación multihilo (y como es de esperar, también por los *plugins* de particionado). Se trata de una aplicación que se podría caracterizar como HPC más que como Cloud.

Presentamos cada uno de los *benchmarks* que vamos a evaluar:

- `in_memory_analytics`: Utiliza Apache Spark para ejecutar un recomendador de películas en base a calificaciones dadas por los usuarios, almacenadas en una base de datos, el cual se ejecuta (junto con su base de datos) en memoria, para evitar las penalizaciones y variaciones del rendimiento provocadas por el acceso a E/S [36].

- `graph_analytics`: También basado en Apache Spark se trata de un *benchmark* de Cloud que realiza análisis de datos sobre grafos. Tiene tres modos de

ejecución (`/pr`, `/cc` y `/tc`, que vamos a evaluar por separado) dependiendo del tipo de recorrido que se hace de los nodos del grafo.

- `pbbcache`: Ya hemos comentado que este *benchmark* no es de Cloud. Además, lo evaluamos dos modos según sus dos modos de funcionamiento: `normal` y `opt`. El modo `normal` realiza simulaciones independientes en cada hilo, mientras que en el modo `opt` se divide el trabajo de una única simulación entre todos los hilos.

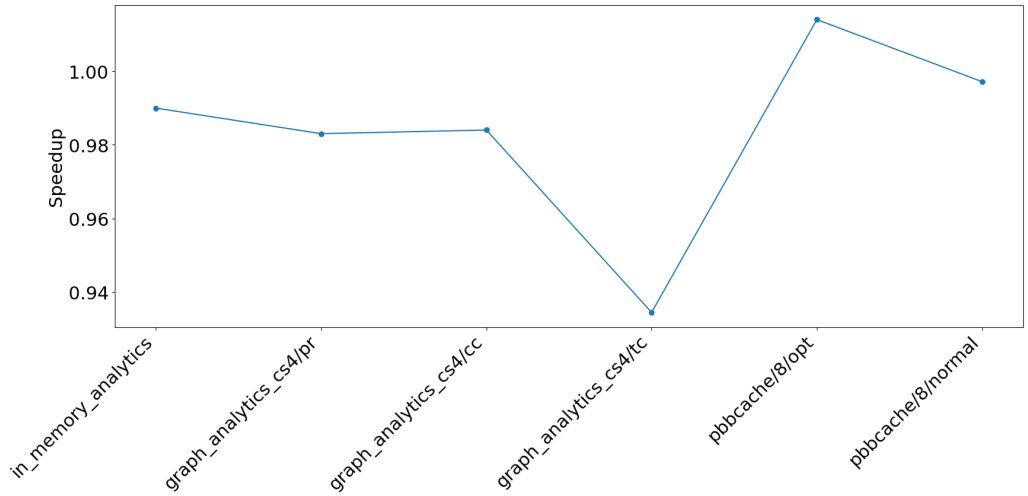

Figura 4.47: Speedup para *benchmarks* de *cloudsuite*.

La figura 4.47 muestra el *speedup* al particionar estáticamente los *benchmarks* anteriores. Como podemos ver sólo `graph_analytics_cs4/tc` empeora significativamente. De nuevo tenemos resultados poco prometedores, que en este caso no vamos a analizar por falta de tiempo.

Capítulo 5

Conclusiones y Trabajo Futuro

Actualmente, los sistemas multinúcleo (CMPs) son una de las arquitecturas más extendidas de propósito general. Es bien conocido que en esta clase de sistemas pueden producirse problemas de rendimiento derivados de la competencia de diferentes aplicaciones (que se ejecutan simultáneamente) por los recursos del sistema que están compartidos entre los distintos núcleos, como es el caso de niveles de caché compartida o el ancho de banda con la memoria principal [5]. Este es el llamado *problema de la contención de recursos compartidos* [6] [7].

Desde hace dos décadas se han venido explorando soluciones a este problema, como *utility-based cache partitioning* (UCP) [10], que, no obstante, requería de un *hardware* especial que no está presente en los procesadores comerciales. Tras la reciente inclusión del soporte hardware para el particionado de caché en procesadores comerciales [11], se han explorado técnicas específicas para aprovechar este soporte, que han logrado mejoras en el rendimiento [13] y la justicia (*fairness*) [12] del sistema.

No obstante, todos estos estudios previos [12] [13] [14] [15] [16] [10] tratan de particionar la memoria caché entre distintas aplicaciones. Por contra, en este trabajo de fin de grado el objetivo era evaluar la idoneidad de particionar la caché entre distintos hilos de una misma aplicación ejecutándose sola en el sistema. Para ello, tras desarrollar el soporte software necesario para utilizar el hardware de particionado Intel CAT, se comenzó evaluando una estrategia de particionado "estático" (que consistía en dividir la caché en particiones iguales y disjuntas, una para cada hilo de la aplicación) sobre una serie de *benchmarks* con el objetivo de identificar aplicaciones propensas a mejorar con el particionado y analizarlas. Inicialmente realizamos esta evaluación en una plataforma equipada con un procesador Intel Alder Lake, un sistema multicore asimétrico de 16 núcleos con soporte de particionado para la caché L2 (pero no para caché L3) en el que sólo los 8 *cores* de bajo consumo comparten caché L2 (una caché L2 compartida por cada 4 *cores*). Los 8 *cores* de alto rendimiento disponen de cachés L2 privadas. Los resultados fueron poco prometedores, pues solo una aplicación (de las 64 evaluadas) mostró una mejora significativa (entendiendo mejora significativa como *speedup* $> 1{,}05$). La conclusión que sacamos de nuestro

estudio en la plataforma Alder Lake es:

- La caché L3 no puede ocultar siempre problemas derivados del particionado de la caché L2. O dicho de otra forma, el particionado (o no) de la caché L2 puede impactar de forma significativa en el rendimiento de una aplicación, a pesar de la caché L3. En la figura 4.4 vimos como el rendimiento de `LU` (una aplicación que realiza una factorización de matrices LU de forma paralela) sufría una importante degradación del rendimiento (*speedup* $\approx 0{,}75$) al particionar estáticamente (dado un conjunto de datos lo suficientemente grande), a pesar de tener una caché L3 no particionada.

Tras llevar a cabo esta ronda de experimentos pasamos a evaluar la estrategia de particionado estático en Broadwell-EP, un sistema simétrico con 8 *cores* que dispone de soporte Intel CAT para particionar la caché L3. Aquí de nuevo los resultados de esta estrategia fueron muy poco prometedores. En definitiva, ninguna aplicación mostró una mejora significativa y solo 4 mostraron una degradación sustancial del rendimiento. Además de medir tiempos de ejecución recopilamos también información de contadores *hardware* durante la ejecución de cada *benchmark* para realizar un análisis de datos del que extraemos las siguientes conclusiones:

- Las aplicaciones cuyos hilos presentan patrones de acceso a memoria heterogéneos tienden a mostrar mayores variaciones en el *speedup* bajo el particionado que las aplicaciones con patrones más homogéneos. Esto es algo que hemos observado continuamente a lo largo del análisis.

- En las aplicaciones cuyos hilos son todos no intensivos en memoria no conviene particionar, pues el acceso a memoria de dichas aplicaciones presenta poco margen de mejora (casi no acceden a memoria) pero se puede producir una degradación enorme del rendimiento, como vimos en `hotspot` y `leukocyte`.

- Enlazando con las anteriores, a pesar de haber evaluado un conjunto muy diverso de *benchmarks*, entre los que se encuentra la *suite* TP-Parsec cuyas aplicaciones son más heterogéneas (no todos los hilos de la aplicación realizan el mismo trabajo) [32], sus patrones de acceso a memoria no lo son, en la mayoría de los casos. Esto provoca que la inmensa mayoría de las aplicaciones no muestren ninguna diferencia significativa entre particionar o no. Concluimos que particionar en el caso general no es una estrategia prometedora.

De cara al trabajo futuro, de nuestro análisis se sigue que algunas aplicaciones heterogéneas podrían beneficiarse de estrategias de particionado más sofisticadas. En *benchmarks* como `backprop` (Rodinia *suite*) o `sort` (BOTS *suite*) se observan mejoras de rendimiento en algunos hilos (redución de tasa de fallos de caché o de accesos totales a memoria) mientras que en otros empeoran, por lo que globalmente no se obtiene beneficio con el particionado estático. De los análisis allí desarrollados se concluye que estrategias de particionado en *cluster* (esto es, las particiones se comparten entre varios hilos) pueden ser prometedoras en algunos casos, ya que el particionado estricto resulta necesariamente en particiones demasiado pequeñas (porque repartimos las vías de la caché, que en nuestra plataforma son 20, entre 8

hilos). En el caso general de particionado de caché entre aplicaciones es bien conocido el efecto beneficioso del *clustering* con respecto al particionado estricto, ya que las pequeñas particiones de este último causan baja asociatividad [12] [13], si bien en el caso de particionado por hilo esto se mitiga parcialmente ya que el particionado solo afecta al reemplazo de líneas, no al acceso a la caché.

En resumen, es de esperar que estrategias más elaboradas de particionado produzcan mejores resultados en algunas aplicaciones concretas, aunque probablemente no en el caso más habitual, donde todos los hilos exhiben un mismo tipo de patrón de acceso a memoria. Particionar por hilo podría ser útil para mejorar el rendimiento de aplicaciones de dominios concretos. En particular que tengan aplicaciones cuyos hilos tienen patrones heterogéneos de acceso a memoria, por ejemplo.

Bibliografía

[1] Carlos Bilbao, *Pmcsched: Scheduling algorithms made easy*, disponible en `https://zildj1an.github.io/pmcsched-website/` (02/08/2024).

[2] Carlos Bilbao, Juan Carlos Saez, and Manuel Prieto-Matias, *Flexible system software scheduling for asymmetric multicore systems with pmcsched: A case for intel alder lake*, Concurrency and Computation: Practice and Experience **35** (2023), no. 25, e7814.

[3] Diego Pellicer and Yikang Chen, *Análisis del potencial del particionado de cache en sistemas multinúcleo para garantizar aislamiento entre aplicaciones y calidad de servicio en la nube*, Trabajo de Fin de Grado en Ingeniería de Computadores (2024).

[4] Intel Corporation, *Intel® 64 and ia-32 architectures software developer's manual. volume 3 (3a, 3b, 3c, & 3d): System programming guide*, 2023.

[5] Vicent Selfa, Julio Sahuquillo, Lieven Eeckhout, Salvador Petit, and María E. Gómez, *Application clustering policies to address system fairness with intel's cache allocation technology*, 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT) (2017).

[6] Chi Xu, Xi Chen, Robert P. Dick, and Zhuoqing Morley Mao, *Cache contention and application performance prediction for multi-core systems*, 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS), 2010, pp. 76–86.

[7] Josué Feliu, Julio Sahuquillo, Salvador Petit, and José Duato, *Understanding cache hierarchy contention in cmps to improve job scheduling*, 2012 IEEE 26th International Parallel and Distributed Processing Symposium, 2012, pp. 508–519.

[8] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto, *Survey of scheduling techniques for addressing shared resources in multicore processors*, ACM Comput. Surv. **45** (2012), no. 1.

[9] Shuang Chen, Christina Delimitrou, and José F. Martínez, *Parties: Qos-aware resource partitioning for multiple interactive services*, Proceedings of the

Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (New York, NY, USA), ASPLOS '19, Association for Computing Machinery, 2019, p. 107–120.

[10] Moinuddin K. Qureshi and Yale N. Patt, *Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches*, 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06), 2006, pp. 423–432.

[11] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer, *Cache qos: From concept to reality in the intel® xeon® processor e5-2600 v3 product family*, 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2016, pp. 657–668.

[12] Juan Carlos Saez, Fernando Castro, Graziano Fanizzi, and Manuel Prieto-Matias, *Lfoc+: A fair os-level cache-clustering policy for commodity multicore systems*, IEEE Transactions on Computers **71** (2022), no. 8, 1952–1967.

[13] Nosayba El-Sayed, Anurag Mukkara, Po-An Tsai, Harshad Kasture, Xiaosong Ma, and Daniel Sanchez, *Kpart: A hybrid cache partitioning-sharing technique for commodity multicores*, 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2018, pp. 104–117.

[14] Mohammad Shahrad, Sameh Elnikety, and Ricardo Bianchini, *Provisioning differentiated last-level cache allocations to vms in public clouds*, Proceedings of the ACM Symposium on Cloud Computing (New York, NY, USA), SoCC '21, Association for Computing Machinery, 2021, p. 319–334.

[15] Sparsh Mittal, *A survey of techniques for cache partitioning in multicore processors*, ACM Comput. Surv. **50** (2017), no. 2.

[16] Lucia Pons, Julio Sahuquillo, Vicent Selfa, Salvador Petit, and Julio Pons, *Phase-aware cache partitioning to target both turnaround time and system performance*, IEEE Transactions on Parallel and Distributed Systems **31** (2020), no. 11, 2556–2568.

[17] Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, Magdalen Dobson, and Yihan Sun, *The problem-based benchmark suite (pbbs), v2*, Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2022).

[18] Juan Carlos Sáez, *Pmctrack*, disponible en `https://pmctrack-linux.github.io/` (26/07/2024).

[19] ———, *Install pmctrack*, disponible en `https://pmctrack-linux.github.io/install/` (26/07/2024).

[20] ———, *Pmctrack gui*, disponible en `https://pmctrack-linux.github.io/gui/` (02/08/2024).

[21] Wikipedia, *Alder lake*, Available at `https://en.wikipedia.org/wiki/Alder_Lake` (08/05/2024).

[22] ———, *Arm big.little*, Available at `https://en.wikipedia.org/wiki/ARM_big.LITTLE` (08/05/2024).

[23] ———, *Broadwell (microarchitecture)*, Available at `https://en.wikipedia.org/wiki/Broadwell_(microarchitecture)` (08/05/2024).

[24] OSDev Wiki, *Model specific registers*, disponible en `https://wiki.osdev.org/Model_Specific_Registers` (02/09/2024).

[25] *A question about the use of highest way in l3_cat*, Available at `https://github.com/intel/intel-cmt-cat/issues/246#issuecomment-1614399852` (08/05/2024).

[26] Intel Corporation, *Best practices for real-time optimizations with the 12th generation intel® core™ processors*, (2022).

[27] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S Fineberg, P Frederickson, T Lasinski, R Schreiber, H Simon, V Venkatakrishnan, and S Weeratunga, *The nas parallel benchmarks*, (1994).

[28] Jaswinder Pal Singh Christian Bienia, Sanjeev Kumar and Kai Li, *The parsec benchmark suite: Characterization and architectural implications*, ACM (2008).

[29] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron, *Rodinia: A benchmark suite for heterogeneous computing*, (2009).

[30] Hamidreza Chitsaz, Raheleh Salari, S. Cenk Sahinalp, and Rolf Backofen, *A partition function algorithm for interacting nucleic acid strands*, Bioinformatics (2009).

[31] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguadé, *Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp*, (2009).

[32] An Huynh, Christian Helm, Shintaro Iwasaki, Wataru Endo, Byambajav Namsraijav, and Kenjiro Taura, *Tp-parsec: A task parallel parsec benchmark suite*, Journal of Information Processing (2018).

[33] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi, *Clearing the clouds: a study of emerging scale-out workloads on modern hardware*, SIGPLAN Not. **47** (2012), no. 4, 37–48.

[34] NASA, *Problem sizes and parameters in nas parallel benchmarks*, disponible en `https://www.nas.nasa.gov/software/npb_problem_sizes.html` (06/09/2024).

[35] Adrian Garcia-Garcia, Juan Carlos Saez, José Luis Risco-Martin, and Manuel Prieto-Matias, *Pbbcache: An open-source parallel simulator for rapid prototyping and evaluation of cache-partitioning and cache-clustering policies*, Journal of Computational Science **42** (2020), 101102.

[36] *In-memory analytics*, disponible en `https://github.com/parsa-epfl/cloudsuite/blob/main/docs/benchmarks/in-memory-analytics.md` (12/09/2024).

Apéndice A

Introduction

Multicore processors constitute today the main architecture choice for general purpose computing systems. Since the cores on a processor aren't truly independent from each other, applications compete for the use of system shared resources, which may lead to substantial performance degradation [5]. This is the so called shared resources contention problem [6] [7]. Performance problems caused by contention issues are unevenly distributed among applications and hard to predict, since they depend on the applications that are co-running on the system and their current phase of execution [8] [9].

Caches are one of those shared resources. While some solutions to shared-cache contention issues were proposed as early as 2006 [10] it was only in the last decade that the main chip manufacturers have finally implemented some hardware support to partition shared-caches on commercial processors [11]. As now hardware support is available some partitioning solutions have been designed to target specific pieces of hardware, for example, Intel CAT [5]. Typically commercially available hardware only support way-partitioning (that is, ways are assigned to partitions) and the number of ways on a cache is usually low. This has caused some clustering strategies to develop, as opposed to strict partitioning of the cache [12] [13]. These techniques allow different applications to share the same cache partition. For example, this allows to isolate applications that don't make an efficient use of cache (they hurt other applications performance when running on a shared cache) by assigning them to the same small partition, thus leaving more cache space to other cache-sensitive applications (applications that benefit from more cache space). On the other side, strict partitioning consists on assigning each application to a different partition. This way applications are fully isolated one from each other. Typically, available cache-partitioning strategies (both clustering and strict partitioning) try to achieve some level of isolation between different applications as a way to improve perfomance and fairness [12] [13] [14] [15] [16] [10]. Thus, on the case of multithreaded applications all threads shared the same partition.

A.1. Goals

The main objective of this project is to explore how to leverage cache partitioning to improve performance of multithreaded applications, when one application of this kind runs alone on the system. Should we find this approach promising, we plan to develop refined cache-partitioning policies that can improve multithreaded applications and containerized (multiprocess) applications performance.

This objective arises from the intuition that, since threads share the same cache, they might (in some situations) remove cache lines that other threads are still using, causing cache pollution that may result in performance degradation.

To complete this main objective we splitted it into minor objetives, which we briefly describe here:

- **Develop per thread cache-partitioning support on the Linux kernel**: We developed plugins for PMCSched (a framework for quick prototyping of scheduling and resource management policies on the Linux kernel) that allow us to easily partition applications at the thread level. The plugins use Intel CAT (Cache Allocation Technology) as the underlying hardware support. We developed a total of two plugins: One for L2 cache partitioning and another one for L3 cache partitioning.

- **Evaluate the potential performance benefit of cache-partitioning at the thread level**: We used the plugins previously developed to evaluate the effect of cache-partitioning at the thread level on performance. To achieve this we used a set of popular HPC (High Performance Computing) benchmarks and measured the performance difference between executing them on a shared cached or on a partitioned cache in which each thread receives the same amount of cache (same number of ways). We refer to this policy "static partitioning". To run these experiments we used Het-Harness, a framework that aids evaluating resource management policies. At this point, we added the PBBS benchmark suite to Het-Harness. Evaluation was performed on two platforms: First, on a desktop machine with Intel Core i9-12900K as processor and Alder Lake microarquitecture which supported L2 cache partitioning. Then on an Intel Xeon CPU E5-2620 v4 machine with Broadwell-EP microarquitecture, this time with L3 partitioning support. From now on we will refer to these machines as "platform Alder Lake" and "platform Broadwell-EP".

- **Hardware counters sampling and analysis**: On this point we sampled hardware counters information using PMCTrack from the benchmarks evaluated before [18]. We calculated some relevant metrics and analyzed the resulting data in order to explain the effect of static partitioning on performance.

A.2. Work plan

This project has been kept on track by holding non-periodic meetings in which the state of current tasks, results or next steps were discussed. In addition to that

asynchronous communication through instant messaging applications and e-mail was used too. Due to the heavy dependencies between tasks the project has had a very linear development. The following tasks were carried out (a timeline is summarized on figure A.1):

T1 Reading several research articles related to cache-partitioning, such as [12], [13] o [10].

T2 Familiarizing myself with PMCTrack and PMCSched.

T3 Learn how to use Het-Harness by reading documentation and attending the introductory meeting.

T4 Read Intel RDT and Intel CAT documentation at Intel's SDM manual.

T5 Implement L2 cache partitioning API on PMCTrack.

T6 Implement L2 cache partitioning plugin on PMCSChed.

T7 Conducting static partitioning experiments on Alder Lake (except PBBS suite).

T8 Add PBBS suite to Het-Harness.

T9 Conducting static partitioning experiments (PBBS suite) on Alder Lake.

T10 Data analysis.

T11 Implement L3 cache partitioning plugin on PMCSched.

T12 Conducting static partitioning experiments on Broadwell-EP (L3 cache).

T13 Collect hardware counters sample during benchmarks execution.

T14 Conducting additional experiments to evaluate static partitioning: Containerized applications.

T15 Writing the report.

A.3. Structure of the report

The rest of this report is structured as follows:

- Chapter 2 describes the tools we used to develop this project, as some familiarity with those tools will be needed to understand the rest of the report.
- Chapter 3 introduces Intel CAT, describes the development of the L2 cache API on PMCTrack and documents the partitioning plugins developed for PMCSched (one for the Alder Lake platform, which partitions the L2 cache and another one for Broadwell-EP which partitions L3 cache).

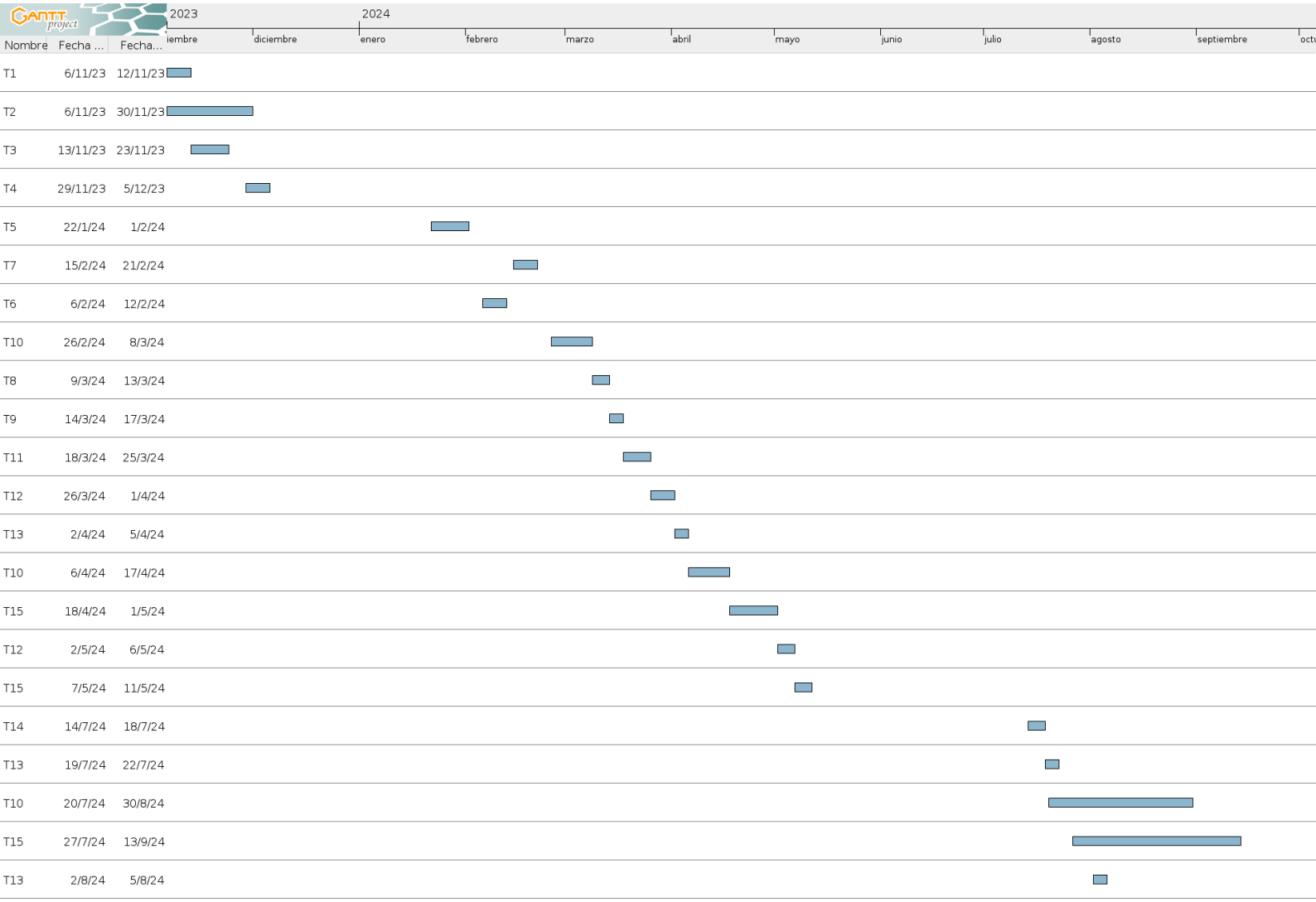

Figura A.1: Gantt diagram

- Chapter 4 showcases our experiments results and includes a long analysis of the metric data obtained from hardware counters on the Broadwell-EP platform.
- Chapter 5 summarizes the main conclusions of this project and outlines several potential future work directions.

Apéndice B

Conclusions and Future Work

Today, multicore systems (CMPs) are the most popular general-purpose arquitecture. It's well known that in this kind of systems competition of different applications (that are co-runners) for shared system resources, such as shared cache levels or main memory bandwidth, may lead to performance degradation [5]. This is the so called problem of contention of shared resources [6] [7].

In the case of shared-caches, solutions have been explored for two decades ago, beginning by the utility-based cache partitioning (UCP). Proposed in 2006 it required some specific hardware that has never been available on commercial processors. After the addition of some hardware support for cache partitioning into commercial processors [11] some partitioning techniques have been developed that target specific hardware. This techniques have proved that it's possible to achieve significant improvements on system performance [13] and fairness [12] by leveraging partitioning hardware.

While previous studies [12] [13] [14] [15] [16] [10] did application-wide partition of the cache, this project goal was to evaluate the potential benefit of partitioning the cache on a per thread basis, in the situation where a single multithreaded application runs alone on the system. To this end, after developing the necessary software support to handle the partitioning hardware (Intel CAT) we began to evaluate the static partitioning policy (which consisted on evenly splitting the cache between threads, having one thread per core on the system) on a set of benchmarks with the objective of identifying applications prone to improve their performance under this policy and subsequently analyze them. At first we did this evaluation on a platform with an Intel Alder Lake processor, a multicore asymmetric system with 16 cores and support for cache partitioning only at the L2 cache, which is private in the case of the 8 fast cores and shared on each socket (1 socket is formed by 4 slow cores) of the 8 slow cores. Results were not promising as only one out of 64 applications showed a significant improvement (we defined significant improvement as a speedup value greater than 1,05). Our conclusion to the evaluation on the Alder Lake platform is:

- The L3 cache can't always hide performance problems derived from L2 cache partitioning. That is, L2 cache partitioning can significantly affect the perfor-

mance of the application, despite the presence of the L3 cache. In the figure 4.4 we saw how the performance of the `LU` benchmark (which performs a LU matrix factorization on a parallel fashion) suffered a significant degradation (speedup $\approx$ 0,75) when doing static partitioning, despite the L3 cache not being partitioned.

After conducting this round of experiments we decided to evaluate the static partitioning policy on Broadwell-EP, a symmetric system with 8 cores and Intel CAT support for L3 cache partitioning. Unfortunately, results on Broadwell-EP weren't promising either, as none of the 64 applications showed significant improvement and only 4 showed a significant performance degradation. In addition to execution times (to calculate speedup) we also collected hardware counters data during the execution of each benchmark, which we later analyzed. From this analysis came the following conclusions:

- The applications whose threads show different memory access patterns have bigger variations on speedup, when static partitioning, than more homogeneous applications. This is something we've noticed several times throughout our analysis on chapter 4.
- In the applications whose threads are all non-memory intensive partitioning causes no benefit, as the time those applications spent accessing memory is very small, and thus there is very little room for improvement but great room for performance degradation to happen (as in the case of `hotspot` and `leukocyte`).
- Despite having evaluated TP-Parsec, a suite of benchmarks with threads that have a more heterogeneous behavior [32], our analysis has shown that their memory access patterns are, in the majority of cases, homogeneous, which results on TP-Parsec applications not showing significant differences under partitioning. Our main conclusion is that partitioning at the thread level is not a promising strategy.

Regarding future work, we have shown on our analysis that some heterogeneous applications could benefit from more sophisticated partitioning policies. Some benchmarks like `backprop` (Rodinia suite) or `sort` (BOTS suite) show improvement on some threads (`llcmpkc` reduction) while other threads worsen (`llcmpkc` increment), which causes the overall application performance to not show a significant difference. We conclude that clustering partitioning policies (policies in which partitions may be shared between threads) might be promising in some cases, as strict partitioning results on too small partitions (we are assigning cache ways, which on our platform are only 20, and must be divided into 8 threads). In the per-application basis, clustering policies have been proven as a better approach than strict partitioning, as the small partitions on the latter case cause low associativity [12] [13] (although in the case of per-thread partitioning this may be somewhat mitigated as the partitioning only applies to cache lines remove, not cache accesses).

To sum up, we expect that more sophisticated per thread partitioning policies may achieve better results on specific applications, although probably not on the general case, where application threads tend to have the same memory access pattern. Per thread partitioning might be useful for applications on specific domains, such as those in which threads have more heterogeneous memory access patterns.