Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Proyecto de Fin de Carrera
Ingeniería Informática
Curso 2013/2014

Caracterización de instrucciones en aplicaciones de cloud

Alba Pedro Zapater

Director: Dr. Víctor Viñals Yúfera

Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Septiembre 2014

Dedicado a mis padres, por estar siempre ahí.

Agradecimientos

Sobre todo quiero agradecer a mi director, Víctor, todo lo que me ha enseñado, su paciencia, sus ánimos, su tiempo, su buen humor y toda la ayuda que me ha proporcionado durante este año. Fue uno de mis mejores profesores en la carrera y sin duda ha sido el mejor director que podría haber tenido. También le doy las gracias a Clemente Rodríguez y a Pablo Ibáñez por el tiempo que me han dedicado y por la gran ayuda que ha sido lo que me han enseñado. Además, darle las gracias a Marta Ortín por estar siempre dispuesta a resolverme dudas y su rapidez para contestarme a los emails.

También a mis compañeros de carrera que me han sorprendido con grandes dosis de solidaridad y ayuda mutua frente a la competencia que mueve nuestro mundo . En especial a Álvaro, Cintia, David y Juan, que han estado siempre para los buenos y los malos momentos y que sin su apoyo y amistad no hubiera sido posible llegar hasta aquí.

Para terminar, quiero agradecerles a mis padres por todo lo que me han dado, sin pedir nada a cambio, tan solo verme feliz. Y a todas las personas que siempre han creído en mi, y de una forma u otra han hecho posible que yo esté ahora escribiendo las últimas lineas de mi proyecto de fin de carrera.

Resumen

Las tendencias de mercado indican que el negocio de los procesadores para grandes centros de datos va a seguir creciendo, impulsado por la economía de la virtualización y la gran penetración empresarial y social de las aplicaciones que residen en las nubes (cloud computing). Para diseñar un procesador de futuro adaptado a este mercado es necesario experimentar con una carga de trabajo apropiada. Por ello, en este proyecto nos hemos centrado en caracterizar el comportamiento de la cache de instrucciones para un sistema de cuatro procesadores, usando el conjunto de aplicaciones Cloudsuite 2.0 del laboratorio de investigación Parsa, representativo del cloud computing.

Hemos usado la plataforma de simulación Simics, un simulador de sistema completo, trabajando con las cinco aplicaciones de Cloudsuite que están acompañadas de checkpoints públicos. Además, se ha contribuido con un tutorial de Simics, acompañado de material práctico, para facilitar y agilizar la fase de formación de otros proyectos que también utilicen esta plataforma.

Para realizar los experimentos deseados se han programado dos módulos de Simics de jerarquía de memoria basados en el módulo g-cache, que implementan dos algoritmos eficientes y específicos para registrar tasas de fallos y huellas de memoria. Un algoritmo obtiene resultados para múltiples caches en una sola simulación y el otro está especializado en caches completamente asociativas.

A partir de estos experimentos hemos analizado los benchmarks en cuanto a su tasa de fallos, en función de su tamaño y de su asociatividad, sugiriendo configuraciones prácticas de tamaño y asociatividad para cada aplicación. También se ha examinado la huella de memoria de instrucciones a lo largo del tiempo, concluyendo que todas las aplicaciones tardan muchos segundos en entrar en régimen estacionario y que la aparición de varias fases complica la selección de ventanas de simulación. Y finalmente, se ha calculado el ancho de banda de instrucciones agregado para los cuatro procesadores simulados, concluyendo que la presión sobre el siguiente nivel puede ser bastante grande, y sugiriendo configuraciones de ese segundo nivel con capacidad para absorber las demandas del primero.

Contenidos

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Las tendencias de mercado indican que el negocio de los procesadores para grandes centros de datos va a seguir creciendo, impulsado por la economía de la virtualización y la gran penetración empresarial y social de las aplicaciones que residen en las nubes (cloud computing).

Para diseñar un procesador de futuro adaptado a este mercado es necesario experimentar con una carga de trabajo apropiada, pero en la actualidad apenas existen programas de prueba (*benchmarks*) de esta clase. Una excepción es la denominada CloudSuite 2.0, un conjunto de aplicaciones cliente/servidor seleccionadas recientemente por el laboratorio de investigación Parsa de la EPFL en Suiza. Estas aplicaciones están pensadas para escalar en un centro de datos de forma horizontal (*scale-out*, es decir, con capacidad para aumentar el rendimiento a medida que se añaden mas computadores independientes) y se caracterizan por su paralelismo explícito y por manejar conjuntos de datos de tamaño muy considerable. Las aplicaciones seleccionadas pretenden ser representativas del futuro del *cloud computing*: Data Analytics, Data Caching, Data Serving, Graph Analytics, Media Streaming, SW Testing, Web Search y Web Serving [EPF].

La experimentación preliminar con estos programas de prueba, publicada en los congresos de arquitectura de computadores, ha revelado un uso intensivo y muy poco eficiente de la jerarquía de cache de instrucciones en chip [FAK+12]. Parece que no solo los conjuntos de datos son muy grandes, sino también el código que los manipula.

Una memoria cache es una memoria RAM estática (SRAM), pequeña y rápida que contiene un subconjunto de las direcciones referenciadas por el procesador. Su funcionamiento es automático, transparente al programador, y se basa en explotar la localidad temporal y espacial del acceso a memoria durante la ejecución de los programas. En los chips actuales de altas prestaciones las memorias cache ocupan una parte sustancial del

silicio, ya que la velocidad en la ejecución de los programas depende en gran medida del rendimiento de las caches. Las memorias cache dentro del chip se organizan como una jerarquía multinivel, ver figura 1.1. El primer nivel de memoria cache está separado en datos e instrucciones para cada procesador; el resto de niveles, hasta dos más, suelen contener de forma mezclada datos e instrucciones. Un buen diseño de la jerarquía de memoria cache permite acceder poco a la memoria principal RAM dinámica (DRAM) situada fuera del chip, contribuyendo de forma crítica a la ejecución eficiente de los programas.

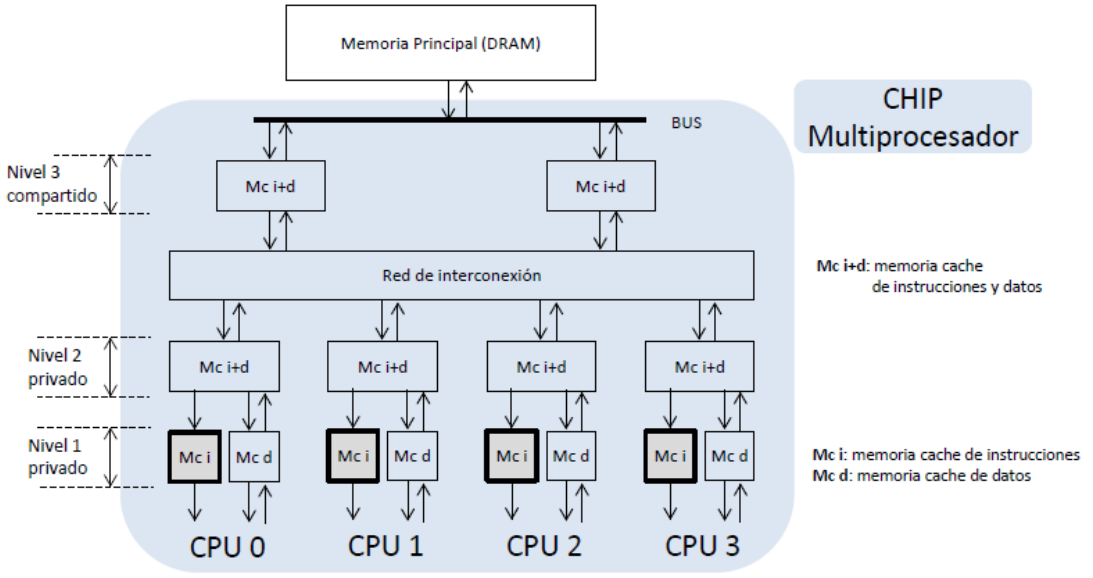

FIGURA 1.1: Ejemplo de chip multiprocesador contemporáneo, con cuatro procesadores y tres niveles de memoria cache en chip (SRAM): los dos primeros privados de cada procesador, y el tercero compartido

En este proyecto nos planteamos una caracterización por simulación del comportamiento de las instrucciones en CloudSuite 2.0. Seguiremos una metodología de experimentación basada en Simics, un hipervisor de tipo 2 con capacidad de emulación de sistema completo: máquinas cliente y servidor con sus periféricos, sistemas operativos huésped (Oracle Solaris 11) y procesadores SPARC v9. Simics no está pensado para desplegar máquinas virtuales orientadas a la consolidación de servidores, sino al desarrollo y prueba de nuevos sistemas hardware y software. Esta orientación permite en nuestro caso configurar un sistema multiprocesador, capturar la secuencia de direcciones de instrucciones e inyectarla a un simulador de memoria cache. Si bien este tipo de simulación de sistema completo (aplicación y sistema operativo) es realmente lenta, la precisión de las conclusiones experimentales es muy elevada.

[Los resultados han sido...]

1.1. Contexto del Proyecto

Este Proyecto Fin de Carrera se ha realizado con el soporte del grupo de investigación en Arquitectura de Computadores de la Universidad de Zaragoza (gaZ) y ha sido financiado en parte por el proyecto TIN2010-21291-C02-01 (Gobierno de España y Unión Europea) y por la dotación anual recibida como grupo consolidado de investigación en Aragón (ref. T48). Además, durante el curso 2013/2014 he disfrutado de una Beca de Colaboración del Ministerio de Educación destinada a la iniciación a la investigación.

1.2. Objetivos

El objetivo de este Proyecto Fin de Carrera es analizar el comportamiento de las instrucciones en CloudSuite 2.0. Para ello se han realizado las siguientes tareas:

1. Instalación y despliegue en cluster de los *checkpoints* necesarios para la simulación con Simics. Un *checkpoint* es un registro del estado de los procesadores, memorias y dispositivos de E/S en un instante dado.
2. Programación de un módulo muy ligero de simulación de cache de instrucciones, que reduce la sobrecarga de las herramientas convencionales de simulación de jerarquía de memoria (p.e. Multifacet GEMS Simulator de la U. de Wisconsin)
3. Análisis de tasas de fallos y de huella de memoria de instrucciones a lo largo de tiempos de ejecución significativos.
4. Conclusiones sobre comportamiento temporal a través de una simulación muestreada y sobre la efectividad de una jerarquía multinivel de instrucciones.

Tras llevar a cabo estas tareas se han alcanzado todos los objetivos inicialmente planteados. El valor añadido en este proyecto se concentra principalmente en los capítulos de resultados y conclusiones, dónde se analizan los resultados obtenidos por simulación.

1.3. Organización de la Memoria

El resto del presente documento está organizado del siguiente modo: en el capítulo 2 se introduce el estado del arte en simulación y cargas de trabajo; en el capítulo 3 se explica con mayor detalle la suite Cloudsuite; el capítulo 4 explica la metodología utilizada para llevar a cabo los experimentos; en el capítulo 5 se presenta un resumen de los resultados

del proyecto y en el capítulo 6 se recogen las conclusiones y las posibles lineas abiertas. Se incluyen como anexos:

- A. Gestión del proyecto. Incluye la planificación del tiempo durante el proyecto y el esfuerzo invertido en el mismo.
- B. Productividad en Simics. Se adjunta un tutorial de Simics que sirva de documentación para futuros proyectos.
- C. Módulo G-cache. Se amplia la información sobre el módulo de g-cache y los módulos programados para el proyecto.
- D. Simulaciones Cloudsuite. Se presenta dónde y cómo se han llevado a acabo las simulaciones de la CloudSuite.

Capítulo 2

Estado del Arte en simulación y cargas de trabajo

La simulación es una herramienta fundamental para diseñar nuevo hardware o mejorar el rendimiento de los programas. En este capítulo describimos brevemente la plataforma Simics y el papel de los programas de prueba en la simulación de nuevas jerarquías de memoria.

2.1. Plataformas y estrategias de simulación

Simics de la empresa Virtutech (simplemente Simics a partir de ahora) es un simulador de sistema completo que podemos configurar para modelar multiprocesadores, sistemas empotrados, routers de telecomunicaciones, clusters o redes de esos elementos [MCE+02]. Es capaz de ejecutar sistemas operativos sin necesidad de que sean adaptados y simular aplicaciones realistas ofreciendo resultados precisos. Se trata de un hipervisor comercial de tipo 2, puede ejecutarse sobre múltiples procesadores y sistemas operativos, y el código no es libre. En la comunidad de experimentación en arquitectura de computadores, Simics suele utilizarse conjuntamente con el entorno GEMS (General Execution-Driven Multiprocessor Simulator) [MSB+05], que fue creado en la Universidad de Wisconsin y proporciona módulos para el estudio de prestaciones en sistemas multiprocesador de memoria compartida con jerarquías complejas y coherentes de memorias cache. El componente principal de GEMS se llama Ruby, que simula las memorias cache, el protocolo de coherencia y la red de interconexión. Simics actúa como un simulador funcional, es decir, simplemente se ocupa de ejecutar las instrucciones, y se comunica con el módulo Ruby de GEMS, que se encarga de gestionar los accesos a memoria, temporizándolos de forma adecuada.

Un problema muy importante en la simulación de multiprocesadores es el bajo rendimiento del simulador, y esto es especialmente cierto en Simics, que como ya hemos dicho es una máquina virtual de sistema completo. Según el detalle temporal (precisión) de la simulación, las aplicaciones se ejecutan entre 100 y 1000 veces más lentas que en la máquina real. En nuestro caso estamos interesados en muy largas simulaciones de fallos/aciertos, sólo para la cache de instrucciones y no nos importa el detalle temporal. Por ello hemos escogido una solución de menos sobrecarga, aunque conllevará la necesidad de un mayor esfuerzo de programación. La solución escogida ha sido el módulo g-cache (que explicaremos en el apartado 4.3), cuyo código fuente acompaña a la distribución estándar de Simics.

2.2. Cargas de Trabajo

Se llama *benchmark* al programa de prueba que sirve para evaluar el rendimiento de un computador completo o de uno de sus subsistemas. [Bon07]. Un conjunto de benchmarks se denomina suite. Históricamente los programas de prueba han evolucionado en complejidad, desde los primeros programas sintéticos, pasando por pequeñas rutinas intensivas en cálculo o memoria, hasta los programas reales de la actualidad, representativos de un campo informático determinado.

La selección de la carga de trabajo tiene una gran importancia, puesto que queremos obtener conclusiones que sirvan para el diseño de los computadores del futuro. Veamos algunas suites actuales:

- **PARSEC 2.1**: suite compilada por la universidad de Princeton (Princeton Application Repository for Shared-Memory Computers, 2009-10). Está compuesta por trece aplicaciones paralelas de memoria compartida (multithreaded applications). Ofrece aplicaciones paralelas típicas, por ejemplo de High-Performance Computing (HPC), pero también incluye otro tipo de aplicaciones paralelas emergentes (p.e. escritorio y servicio WEB). Recoge distintas dominios de aplicación, como visión por computador, codificación de vídeo, análisis financiero, visualización de experimentos físicos y proceso de imagenes.[BKSL08]
- **SPEC CPU2006**: suite compilada por la cooperativa SPEC en 2006 (Standard Performance Evaluation Corporation). Está pensada para medir el rendimiento del procesador, la jerarquía de memoria o el compilador, puesto que no tiene apenas operaciones de entrada/salida. Contiene dos suites de benchmarks, una intensiva en cálculo entero (12 programas no paralelos) y otra en coma flotante (19 programas no paralelos). [CPU06]

- **SPECweb 2009**: también de la cooperativa SPEC, busca evaluar el rendimiento de servidores WEB. Sus cargas de trabajo están pensadas para multiprocesadores de memoria compartida e incluyen aplicaciones de banca, comercio electrónico o soporte Web.[web09]
- **TPC-C**: aplicación patrocinada por la cooperativa Transaction Processing Council desde 1992. En la actualidad está en su versión cinco, y simula un entorno completo de usuarios realizando transacciones en directo (online) hacia una base de datos. Aunque no se limita a ninguna actividad en particular modela una empresa que debe gestionar, vender y/o distribuir un producto o servicio.[TC]

Los anteriores benchmarks son un buen resumen de aplicaciones que se están ejecutando en los computadores actuales. Sin embargo, unos nuevo tipo de aplicación está emergiendo con fuerza en los últimos años: los servicios de la nube (cloud computing). Esta plataforma está dominando el suministro de servicios escalables online. Estos servicios se caracterizan por unos enormes *working-sets*, un alto grado de paralelismo y restricciones de tiempo real no estricto. Todo esto hace que estas aplicaciones denominadas *scale-out* tengan un comportamiento distinto a las aplicaciones tradicionales ya conocidas y que se recogen en los benchmarks anteriores. Por ello, para estimular la investigación en el área de los centros de datos y la nube y ya que apenas existen benchmarks de esta clase, el laboratorio de investigación Parsa de la EPFL en Suiza ha creado CloudSuite, un benchmark basado en servicios online del mundo real [FAK+12]. Esta es la carga de trabajo que hemos seleccionado para nuestro proyecto, por lo que explicamos sus características con más detalle en el capítulo siguiente.

Capítulo 3

CloudSuite

Como ya hemos introducido en el capítulo anterior, la carga de trabajo que usamos en este trabajo es Cloudsuite 2.0, un conjunto de aplicaciones cliente/servidor del grupo de investigación Parsa de la EPFL en Suiza. Estas aplicaciones están pensadas para escalar en un centro de datos de forma horizontal (*scale-out*: a más servidores físicos, más rendimiento) y se caracterizan por su paralelismo explícito y por manejar conjuntos de datos de tamaño muy considerable. Aunque en su web podemos encontrar disponibles 8 aplicaciones para ejecutar en nativo [EPF], nosotros hemos trabajado solo con 5, aquellas acompañadas de checkpoints públicos para la simulación en Simics. En la tabla 3.1 podemos ver las aplicaciones con una breve descripción:

Aplicación	**Descripción**
Data Analytics	Esta aplicación se basa en el paradigma map-reduce, que ha emergido como una aproximación muy popular para los análisis de datos a gran escala. Se lanzan peticiones al cluster de procesadores que se simulan, que en primer lugar filtran y transforman la información (map) y después unen los resultados (reduce).
Data Serving	Aplicación de almacenamiento y servicio de datos basada en NoSQL (Not only SQL). Ha sido diseñada explícitamente para soportar aplicaciones web como Facebook, Google Earth y Google Finance, proporcionando almacenamiento escalable, con capacidad de adaptar rápidamente el esquema de almacenamiento.
Media Streaming	Los servicios en streaming, tipo Youtube, usan enormes clusters de servidores que gradualmente empaquetan y transmiten ficheros multimedia cuyo tamaño puede ir desde los megabytes hasta los gigabytes.
Web Frontend	Las aplicaciones que dan servicio al alojamiento de páginas web se caracterizan por su gran tolerancia a fallos y su escalabilidad dinámica.
Web Search	Aplicación basada en un motor de búsqueda, similar a Google, capaz de indexar terabytes de datos recogidos dinámicamente de fuentes online.

Tabla 3.1: Aplicaciones CloudSuite 2.0 simuladas.

3.1. Características de los Benchmarks

Todas estas aplicaciones tienen unas características similares [FAK+12]:

- Operan con grandes conjuntos de datos que se reparten entre un gran número de máquinas, típicamente en fragmentos residentes en las memorias principales de los servidores.
- Sirven grandes cantidades de peticiones completamente independientes que no comparten ningún estado.
- Están diseñadas específicamente para una infraestructura de servidores típica de la nube, donde las conexiones y las máquinas no son del todo fiables.
- Usan conectividad entre máquinas solo para las tareas más importantes de coordinación y administración.

En algunas aplicaciones de esta suite llegamos a simular hasta tres computadores completos conectados por red, como es el caso de Web Frontend, cuyo esquema está representado en la figura 3.1. Este benchmark consiste en tres componentes principales: el servidor web, la base de datos y un cliente, cada una es ejecutada en una máquina distinta y emulan los accesos del mundo real al servidor web. Todo este sistema simulado nos permite estudiar las instrucciones del servicio crítico: el servidor web que se ejecuta en una máquina multiprocesador, de cuatro procesadores en nuestro caso.

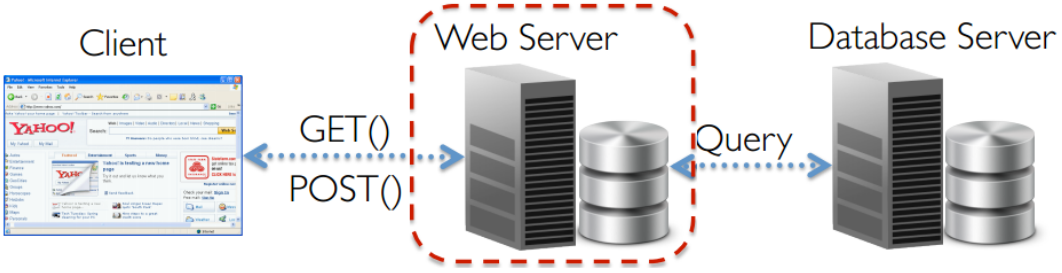

FIGURA 3.1: Esquema Aplicación Web Frontend.

3.2. Cloudsuite en Simics

Como ya se ha apuntado, únicamente se disponen de forma pública los checkpoints para Simics de las cinco aplicaciones de la tabla 3.1. Los checkpoints permiten empezar una simulación en un punto de interés, sin necesidad de configurar todo de nuevo, arrancar la máquina, y saltar la fase de inicialización. Un checkpoint almacena el contenido de los registros de los procesadores, de las MMUs, la imagen de la memoria principal, los contenidos de los discos y el estado de los periféricos (consola, conexiones de red, etc.). En nuestro caso un checkpoint consiste en varios ficheros que contienen la configuración del sistema simulado (máquinas para los clientes, para la base de datos y para el servidor bajo análisis) en un estado estacionario de la ejecución, saltando la fase de inicialización del sistema que queremos analizar. Para desplegar los checkpoints en ATPS, nuestro cluster de experimentación, ha sido necesario configurar las rutas que referencian a los diferentes ficheros de un checkpoint: datos de entrada de la aplicación simulada, configuración hardware de las máquinas, e imagen del estado hardware en el punto de inicio de la simulación.

Capítulo 4

Metodología

Este capítulo recoge la metodología utilizada durante el proyecto. Se presentan las métricas seleccionadas, las herramientas que se han elegido para obtenerlas y cómo se han usado.

4.1. Métricas Utilizadas

Estamos interesados en estudiar el rendimiento de la cache de instrucciones, en un sistema multiprocesador de memoria compartida, centrándonos en la máquina que ejecuta el servicio crítico en las cinco aplicaciones seleccionadas de CloudSuite 2.0. El sistema de interés es el representado en la figura 4.4.

Como se verá en el capítulo siguiente, para analizar el comportamiento durante un tiempo significativo, hemos optado por obtener estadísticas de forma periódica. El conjunto de métricas que obtenemos en cada muestra temporal, para cada procesador, forma una traza temporal que almacenamos para manipulaciones posteriores. De esta forma, podremos estudiar la variación en el tiempo o calcular un agregado, según la escala temporal que nos interese considerar.

A continuación describimos las métricas y modelos que vamos a usar para analizar el cómo se referencian las instrucciones y obtener, si es posible, conclusiones de diseño.

- **Mpki** : Número medio de fallos de la cache de instrucciones por cada mil instrucciones ejecutadas (*Misses per kilo instruction*), en este trabajo nos centramos de la cache de instrucciones de primer nivel. Recordemos que cada vez que se produce un fallo, entra el bloque de cache requerido, de 64 bytes.

- **BWin**: Ancho de banda de instrucciones (*Instruction Bandwidth*) entrante desde el siguiente nivel. Se agrega para las cuatro caches. Es el cociente entre el número de bytes de instrucciones entrante a las caches y el tiempo de ejecución.
- **Huella de memoria de instrucciones**: (*Instruction Footprint*) Tamaño del programa referenciado durante la ejecución, en número de bytes. Se trata de contar instrucciones *diferentes*. Por ejemplo, un bucle de diez instrucciones que se repite 1000 veces supone una huella de 40 bytes (10 instr. x 4 bytes/instr.)

4.2. Modelo de las 3C

El "modelo de las 3C"(*compulsory, capacity, and conflict*) es uno de los más usados y conocidos para el estudio de los fallos en las memorias caches [HS89]. Para entender el modelo, en la figura 4.1 se presenta una memoria cache genérica, con los parámetros de diseño a tener en consideración.

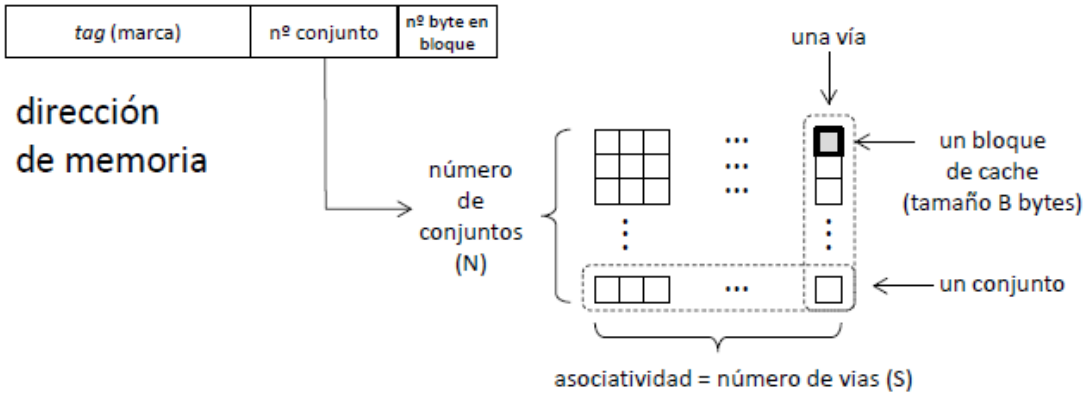

FIGURA 4.1: Memoria cache genérica de tamaño NxSxB bytes. Se escogen $log_2 N$ bits de la dirección para indexar conjunto. En caso de fallo, si el conjunto correspondiente está lleno, se reemplaza al bloque menos recientemente usado (LRU).

El modelo se basa en tres tipos de fallos:

- **Obligatorios (Compulsory)**: Son los producidos por la primera referencia de un bloque en memoria. Este número no se ve afectado por la asociatividad o el tamaño de la memoria cache. Estos fallos corresponderían a los fallos que tendría una cache completamente asociativa (un solo conjunto) de tamaño infinito, y contabilizan el número total de bloques que se cargan desde el siguiente nivel.
- **De Capacidad (Capacity)**: Son aquellos fallos que se dan en una cache completamente asociativa con política de reemplazo LRU (*least recently used*), *menos* los fallos obligatorios. Estos fallos aparecen por referenciar mayor cantidad de bloques

que los que caben en la memoria, por lo tanto estos fallos dependen del tamaño de la cache.

- **De Conflicto (Conflict)**: Son los fallos totales de la cache *menos* los de capacidad y los obligatorios. Son dependientes de la asociatividad, ya que corresponden aquellos fallos que se dan por tener que alojar bloques en el mismo conjunto.

Para representar este modelo se obtienen los fallos para distintos tamaños de cache con distintas asociatividades, incluyendo siempre la completamente asociativa y la correspondencia directa (o asociatividad 1).

En las gráficas 4.2 y 4.3 vemos un ejemplo de las dos representaciones habituales del modelo para la cache de instrucciones de la aplicación Media Streaming con un único procesador, los primeros 4'5 segundos de ejecución y con 32B de tamaño de bloque. Ambas gráficas muestran fallos en función de la capacidad, desde 2 hasta 2048 KB en el eje X. El eje Y representa tasa de fallos en mpki o porcentaje relativo de cada tipo de fallos, respectivamente. En la primera gráfica se observa como las tasas de fallos disminuyen desde los 15-17 mpki para 2KB, hasta una cifra inapreciable para 2 MB. Para 64 KB, por ejemplo, podemos ver cual es la penalización por disminuir asociatividad: pasamos progresivamente de los 2,57 mpki en completamente asociativo, hasta los 5,6 mpki de correspondencia directa. En la gráfica 4.3 vemos los porcentajes relativos; para la cache de 64 KB los fallos obligatorios suponen una porcentaje no apreciable, mientras que al disminuir la asociatividad (desde completamente asociativo hasta S=1, pasando por S=4 y 2), los fallos suponen el 46 %, 65,9 %, 89,5 % y 100 %, respectivamente. Dicho de otra forma, manteniendo el tamaño fijo a 64 KB, si cambiamos la organización de la cache y pasamos de S=1 (N = 1024 conjuntos) a S=2 (N = 512 conjuntos), los fallos bajan aproximadamente un 10 %.

La gráfica 4.2 ilustra una anomalía que no suele aparecer en caches de datos y a veces se observa en caches de instrucciones. Vemos que en la cache de 32KB hay más fallos para asociatividad 4 que para asociatividad 2. En principio la intuición dice que a mayor asociatividad, menos fallos de conflicto, y menos fallos totales (por cierto, a mayor asociatividad una cache requiere mas energía por acceso y resulta en un mayor tiempo de acierto). Sin embargo, en el punto reseñado no es así, ¿porqué?. Consideremos un ejemplo extremo: dos caches de T bloques, una de correspondencia directa (S=1) y otra completamente asociativa (N=1). Supongamos un bucle de instrucciones cuyo tamaño supera *en un bloque* al tamaño de la cache, es decir, T+1 bloques. En régimen permanente la cache de correspondencia directa se carga con T-1 bloques que no se mueven, pero hay un conjunto al que van a parar dos bloques en cada iteración; el resultado es una tasa de fallos muy baja: 2 fallos cada Tx16 referencias (64 Bytes = 16 instrucciones). En

cambio, en la cache completamente asociativa, debido al algoritmo de reemplazo LRU, el régimen permanente resulta en T fallos cada Tx16 referencias, ya que las últimas referencias sobrescriben a las primeras, las primeras a las segundas y así sucesivamente ...

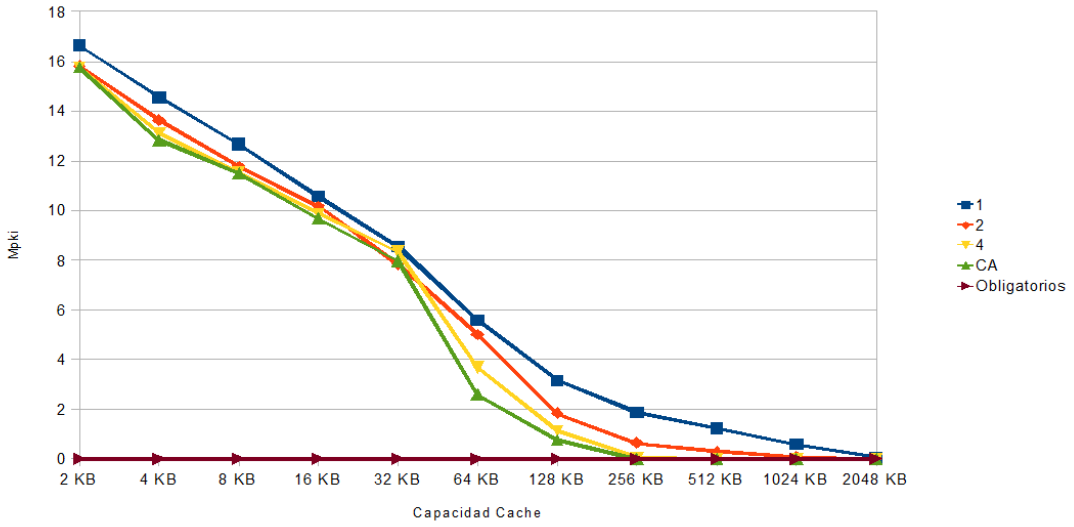

FIGURA 4.2: Ejemplo modelo de las 3C con mpki. Aplicación Media Streaming, un procesador, 4,5 segundos de ejecución.

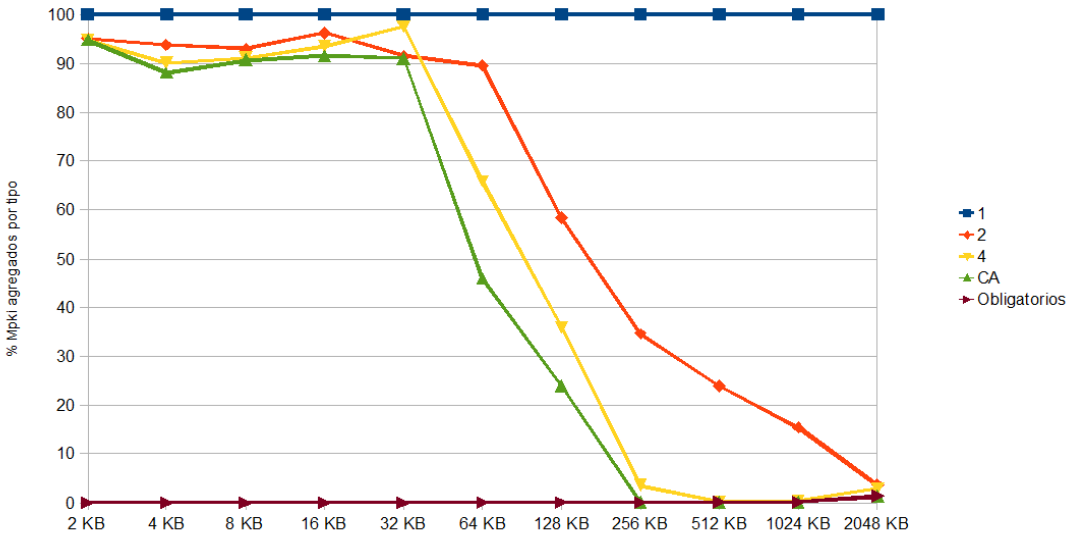

FIGURA 4.3: Ejemplo modelo de las 3C con porcentajes agregados. Aplicación Media Streaming, un procesador, 4,5 segundos de ejecución.

En nuestro proyecto hemos simulado un sistema de cuatro procesadores tal como se puede ver en la figura 4.4. Para caracterizar caches en sistemas multiprocesador puede usarse el modelo ampliado de las 4C [CGS99]. La cuarta fuente de fallos (*coherence*) proviene de las invalidaciones necesarias para mantener la coherencia, sin embargo este tipo de fallos no aparece en el flujo de instrucciones, que no están sometidas a escrituras compartidas.

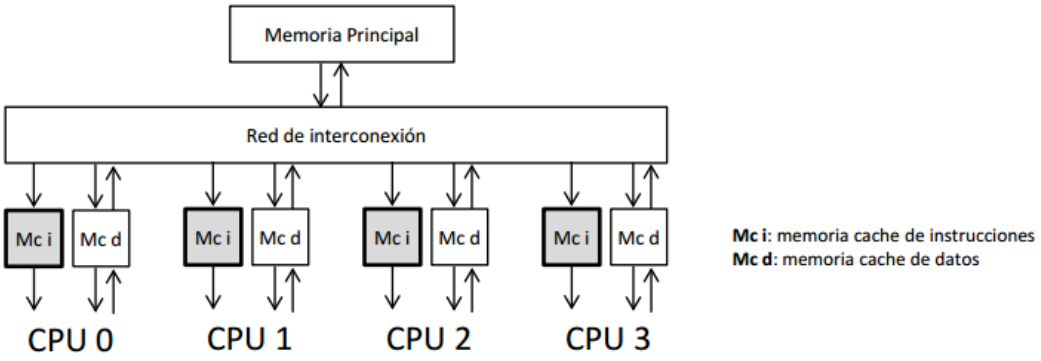

FIGURA 4.4: Sistema de cuatro procesadores simulado. Las memorias cache de instrucciones han sido modeladas en detalle.

4.2.1. Algoritmos de una sola pasada

Para construir las gráficas 4.2 y 4.3 podemos ejecutar una simulación por cada tamaño de cache y por cada asociatividad. Sin embargo este procedimiento requiere mucho tiempo de simulación. Alternativamente, podemos usar un algoritmo "de una sola pasada" que nos permite en una sola simulación obtener los fallos para distintas asociatividades y tamaños de cache [MGST70]. Estos algoritmos son factibles para políticas de reemplazo de tipo pila, para los cuales aumentar la asociatividad manteniendo el número de conjuntos siempre resulta en una tasa de aciertos mayor. La política de reemplazo LRU (Least Recently Used) cumple esta condición. LRU precisa una pila ordenada en cada conjunto. En la cima de la pila está el bloque mas recientemente utilizado (MRU, Most Recently Used), y en el fondo el menos recientemente utilizado (el LRU), es decir el bloque que será víctima en caso de reemplazo. En la figura 4.5 se ilustran los casos de acierto y fallo a un conjunto cualquiera. Se puede ver el estado original de la pila LRU del conjunto, y su nuevo estado al llegar una referencia al bloque C. Cómo ya está en el conjunto se produce un acierto a distancia 3 en la pila y este bloque pasa a ser el MRU. La siguiente referencia que llega es al bloque E, que no está en la cache, por lo tanto se

produce un fallo. El bloque víctima, el que se expulsa, será el LRU, es decir el bloque D. Y el bloque E queda en la cima de la pila, es decir la posición MRU.

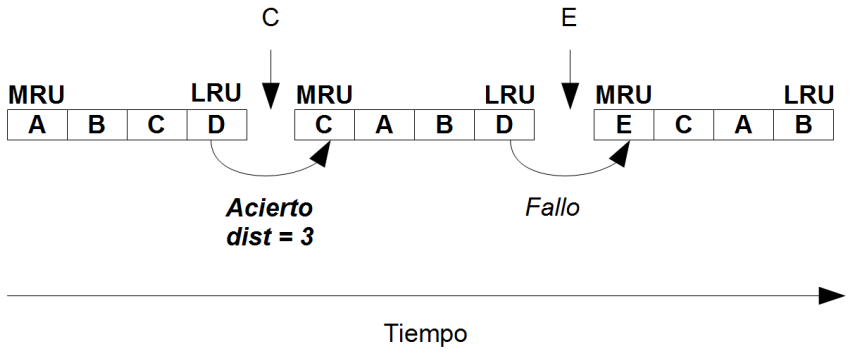

FIGURA 4.5: Pila de bloques en orden LRU en un conjunto cualquiera. Acierto en el bloque C seguido de fallo del bloque E.

Analizando la pila ordenada de bloques de un conjunto cualquiera, vemos que el acierto *a distancia 1* de la cima (la propia cima) se produce si un bloque vuelve a referenciarse inmediatamente. El acierto *a distancia 2* se produce si entre dos referencias al mismo bloque se referencia uno distinto. Aplicando este razonamiento de forma sucesiva se puede demostrar que, manteniendo fijo el número de conjuntos, una cache de asociatividad S experimenta los mismos o mas aciertos que una cache de asociatividad S-1, ya existe una relación de *inclusión* entre los contenidos de la cache de asociatividad S y la cache de asociatividad S-1. Esta relación es más fácil de apreciar a través de un ejemplo, para ello vamos a ayudarnos de las figuras 4.6 y 4.7. En ambas el tiempo transcurre de izquierda a derecha. Entre estado y estado aparece en la parte superior cual es el bloque referenciado y en la parte inferior si se produce acierto o fallo. En el caso de acierto se apunta también a que distancia de la cima de la pila LRU se ha producido.

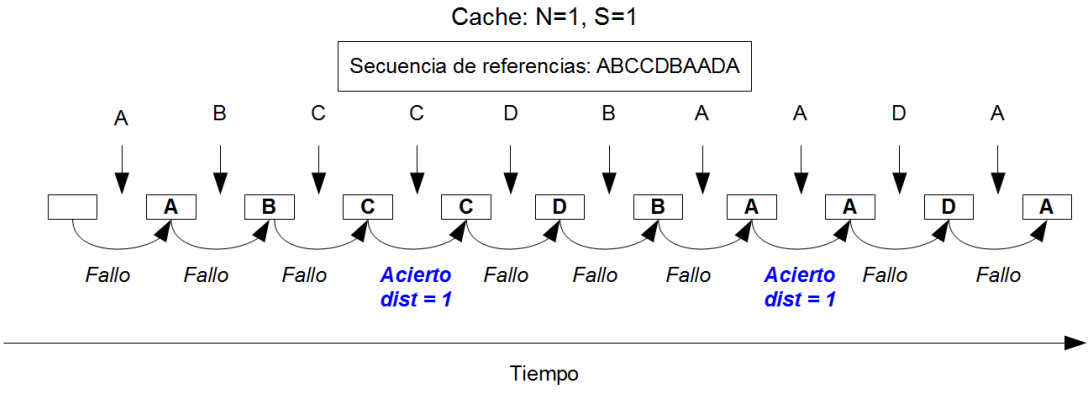

FIGURA 4.6: Aciertos y fallos para una cache LRU con asociatividad 1

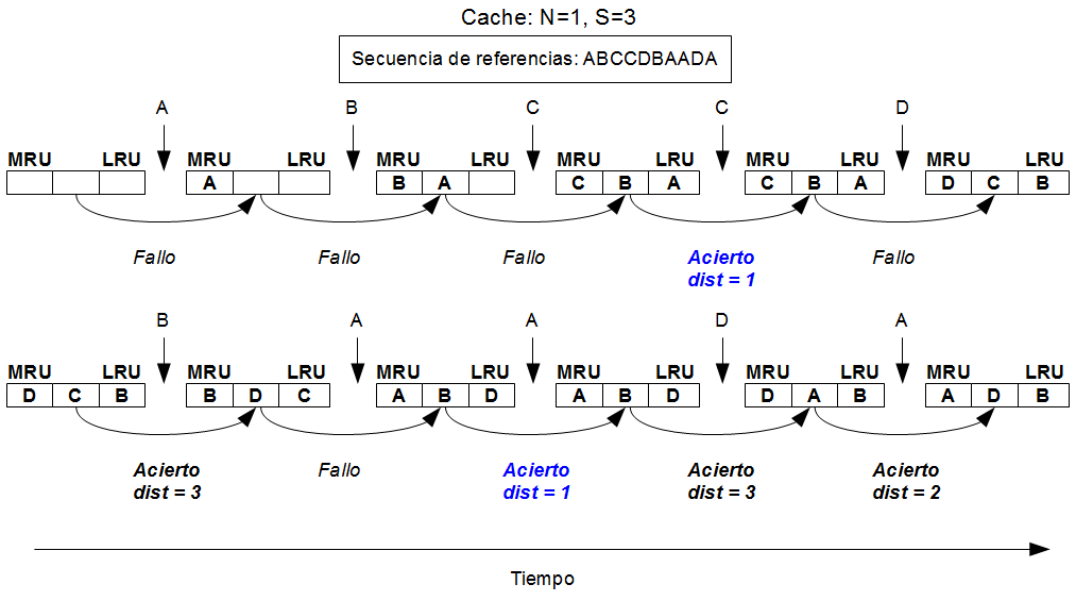

FIGURA 4.7: Aciertos y fallos para una cache LRU con asociatividad 3

En la primera figura, la 4.6, se representa una cache de un solo bloque, y de un solo conjunto, así que si inmediatamente no se referencia al mismo bloque, se produce un fallo. En el caso de que se produzca un acierto este siempre es a distancia 1, ya que la pila LRU del conjunto solo tiene un elemento. En el vector de la izquierda de la figura 4.8 podemos ver el resumen de los aciertos y fallos totales de la secuencia de referencias utilizada (ABCCDBAADA).

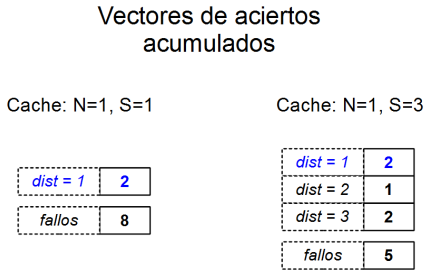

FIGURA 4.8: Vector de aciertos acumulados para S=1 y S=3

Sin embargo la secuencia de estados en la figura 4.7 se complica, ya que corresponde a una cache de un solo conjunto, pero de asociatividad 3. Ahora aparecen aciertos a distancias 1, 2 y 3. Aquí se aprecia lo que explicábamos anteriormente, por ejemplo, el último acierto se da a distancia 2 porque entre la última referencia al bloque A y la anterior referencia solo se ha referenciado al bloque D. O en el penúltimo acierto, que se da a distancia 3 porque entre la última vez que se referencia a D y la anterior sólo se han

referenciado dos bloques: el A y el B. En el vector de la derecha de la figura 4.8 recogemos el número total de fallos y de aciertos, estos últimos clasificados por distancias.

Gracias a las figuras y a las tablas ahora podemos ver mejor porque los aciertos de asociatividad 1 de una cache están incluidos en los aciertos de una cache con el mismo número de conjuntos pero mayor asociatividad. Los aciertos a distancia 1 del vector S=3 en 4.8 son los correspondientes a los aciertos de la cache de asociatividad 1 de la figura 4.6 para esa secuencia de referencias a bloques.Y llegamos a la conclusión de que no es necesario representar una cache de asociatividad 2 con 2 bloques, y un solo conjunto, para saber su número de aciertos ya que la cache de asociatividad 2 incluirá los de la 1 (2 aciertos) más los de distancia 2 del vector S=3 en 4.7 de la cache de asociatividad 3. Es decir, para la cache de un conjunto y asociatividad 2 el número de aciertos será 3, y por lo tanto el número de fallos 7, ya que en total hay 10 referencias a bloques.

Asumiendo reemplazo LRU, ¿cómo concretar estas ideas en un algoritmo?. Una forma es gestionar un *vector de aciertos* que contabiliza cuantos aciertos se dan en cada distancia. El algoritmo en detalle puede consultarse en el anexo C.2.1.

Con este algoritmo podemos obtener, por ejemplo, a partir de la simulación de una cache de 16KB de asociatividad 4, los aciertos (y por lo tanto también los fallos) de una cache 8KB con asociatividad 2, y de una cache de 4KB con asociatividad 1.

4.3. Módulo G-Cache

4.3.1. Módulos en Simics

Un módulo en Simics es un código ejecutable que se carga dinámicamente en la máquina virtual. Para tener un uso práctico debe interactuar con Simics, con otros módulos o con el usuario. Simics proporciona una API (*application programming interface*) para que los módulos puedan utilizar diversas funciones. La API soporta los conceptos de clase, objeto, interfaz y evento. Los módulos pueden programarse en DML (Device Modeling Language), Python o C/C++.

En este proyecto hemos trabajado modificando un módulo ya definido por Simics, g-cache, que se explica a continuación.

4.3.2. G-cache

Simics es una máquina virtual con capacidad de ejecución funcional de sistema completo, tanto de aplicaciones como de sistema operativo. Por tanto no modela las cuestiones de

implementación transparentes al lenguaje máquina, como la jerarquía de caches. Sin embargo, incorpora a modo de ejemplo el módulo g-cache que permite modelar una jerarquía multinivel de caches para multiprocesador. G-cache trata las transacciones de memoria de forma simple: todas las operaciones necesarias (*copy-back* de bloques sucios de datos, *fetch* de instrucciones, etc.) se ejecutan en orden de programa y una sola vez. La cache devuelve la suma de los ciclos de parada para cada operación. Hemos modificado este modulo para programar de forma eficiente nuestras caches de instrucciones. Las dos versiones programadas tienen la siguiente funcionalidad:

- **Algoritmo para múltiples caches (algoritmo MC)**: Se aplica la idea del apartado 4.2.1 para recoger en una simulación única los fallos de varias asociatividades y tamaños.
- **Algoritmo para caches completamente asociativas (algoritmo CCA)**: Ya que las caches completamente asociativas solo tienen un conjunto, los algoritmos tradicionales de reemplazo LRU es muy costoso de simular para caches grandes, por tener que recorrer toda la lista LRU una o varias veces cada vez que se produce un fallo. La mejora original que proponemos es utilizar una cache de correspondencia directa auxiliar, que permite capturar una gran parte de los aciertos, evitando tener que buscar el bloque en la cache simulada. Hemos medido una mejora media en velocidad de un 90,52 % gracias a esta mejora.

En el Anexo C se describen en detalle los dos algoritmos.

4.4. Experimentos

En este trabajo hemos lanzado 6 experimentos con el algoritmo MC, y 5 experimentos con el algoritmo CCA, por cada aplicación. La gráfica 4.9 muestra los seis primeros experimentos y a qué cache (asociatividad y tamaño) corresponden los resultados obtenidos, cuatro caches distintas por cada experimento. Así que con seis ejecuciones hemos obtenido los datos de 24 caches distintas, suponiendo una muy importante mejora.

El algoritmo CCA se ha ejecutado para las caches de 16KB, 32KB y 64KB, para las que ya disponemos resultados desde asociatividad 1 a 8, pudiendo así completar el modelo de las 3Cs.

El cuarto experimento con este algoritmo corresponde a la simulación de una cache de 2048KB. Esta cache que al ser lo suficientemente grande nos permite contabilizar los fallos obligatorios.

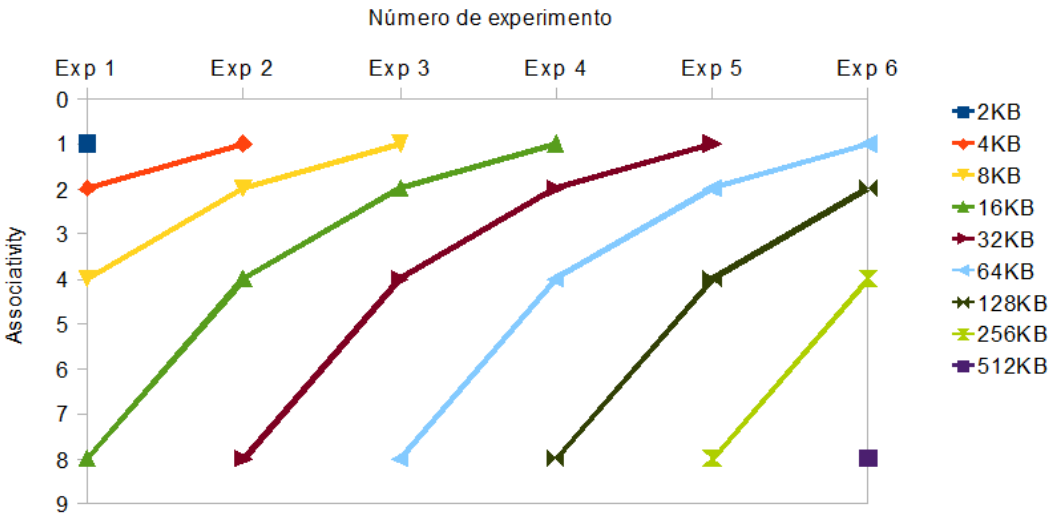

FIGURA 4.9: Experimentos lanzados con algoritmo MC.

Cada uno de los experimentos anteriores supone 10 segundos (tiempo en máquina real) de cada aplicación, recogiendo muestras cada 100ms, es decir se obtienen 100 muestras por cada experimento y aplicación. Antes de cada muestra las caches y sus estadísticas se inicializan.

Por último, el quinto experimento con el algoritmo CCA corresponde a la obtención de los fallos obligatorios a lo largo de los 10 segundos (tiempo en máquina real) sin inicializar las caches ni las estadísticas y recogiendo los datos cada 100ms . El experimento se ha lanzado con un tamaño de 2048KB para todas las aplicaciones excepto para Classification y Cloudstone que ha sido de 4096KB y 8192KB,respectivamente.

Capítulo 5

Resumen de Resultados

En este capitulo se recogen los resultados de los experimentos realizados, siguiendo las métricas presentadas en el apartado 4.1.

Los resultados se estructuran en tres apartados; el primero muestra las tasas de fallos promediadas para todos los cores en toda la duración de las aplicaciones; el segundo presenta la huella de memoria, estudiando su evolución temporal en intervalos de 100ms; el tercer apartado presenta el ancho de banda agregado que debe suministrar el siguiente nivel, también analizando intervalos de 100 ms. Al final de cada apartado se ofrecen unas conclusiones de comportamiento y, en su caso, de diseño.

5.1. Mpki por Core

Las gráficas presentadas en esta sección (5.1 - 5.5) resumen el comportamiento de las caches de instrucciones en cuanto a su tasa de fallos expresada en mpki, en función de su tamaño y de su asociatividad, para un tamaño de bloque de 64 bytes. Para cada tamaño de cache y para cada core hemos calculado la media aritmética de todas las muestras temporales[1].

En estas gráficas observaremos la importancia relativa del tamaño y la asociatividad en la tasa de fallos, así como la posible diferencia de comportamiento entre cores.

[1]Estos datos, junto con los del siguiente apartado, permiten descomponer los fallos según el modelo de las 3Cs. No se ha hecho así porque la anomalía de asociatividad aparece, y entonces la representación pierde utilidad.

5.1.1. Streaming (Figura 5.1)

Destaca la diferencia de comportamiento entre cores: el core 3 es menos sensible a la asociatividad y al tamaño (rango total 35-20 mpki), mientras que los cores 1,2 y 4 tienen comportamientos casi idénticos, presentando unas tasas altas para 16KB (45-50 mpki) y mucha sensibilidad a la asociatividad para 64 KB.

En los dos grupos de cores aparece la anomalía de asociatividad. Para el core 3, en 64 KB solo la correspondencia directa es peor que completamente asociativo. Para el resto de cores ocurre algo muy parecido, pero para 32 KB.

Resaltemos esto: la mejor elección de asociatividad se invierte por completo, según el tamaño y el core considerados, lo cual no es nada bueno desde el punto de vista de diseño. Escoger una asociatividad 4-8 para todos los cores y tamaños, podría ser un buen compromiso de diseño.

En definitiva, estamos frente a una aplicación cuya búsqueda de instrucciones puede convertirse en el cuello de botella del procesador si el tamaño de cache es insuficiente o la asociatividad no es la apropiada.

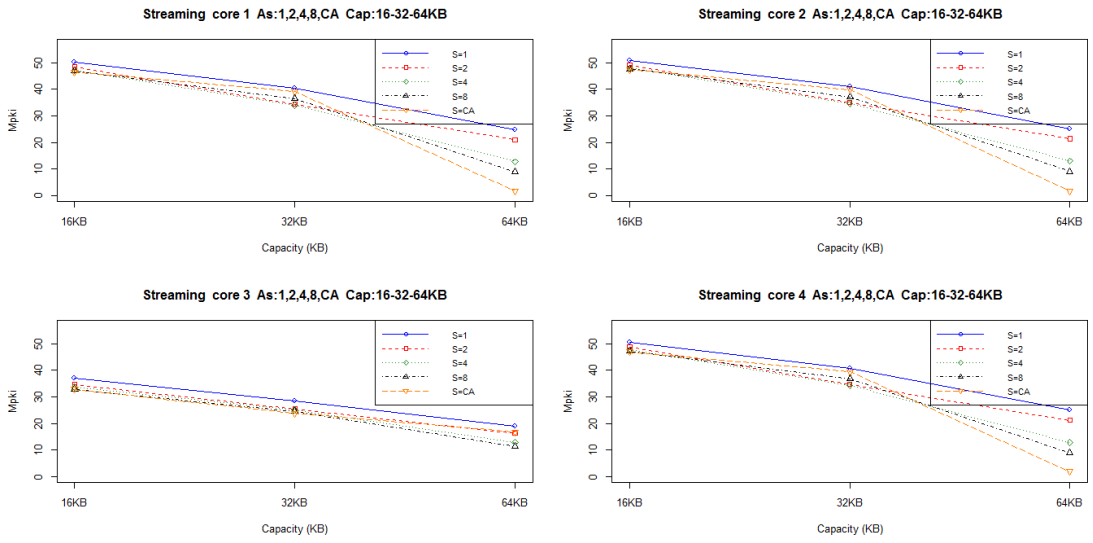

FIGURA 5.1: Streaming: mpki de la cache de instrucciones en cada core vs. tamaño y asociatividad. Tamaño de bloque 64B.

5.1.2. Cassandra (Figura 5.2)

En esta aplicación basta con descartar diseños de correspondencia directa para obtener un muy buen rendimiento (4-5 mpki), independientemente del tamaño y del core considerado. En cuanto a comportamiento de cache, parece que esta aplicación paralela

usa el mismo código en los cuatro procesadores. No se observa ninguna anomalía de asociatividad, y la sensibilidad de los fallos al tamaño de cache es reducida.

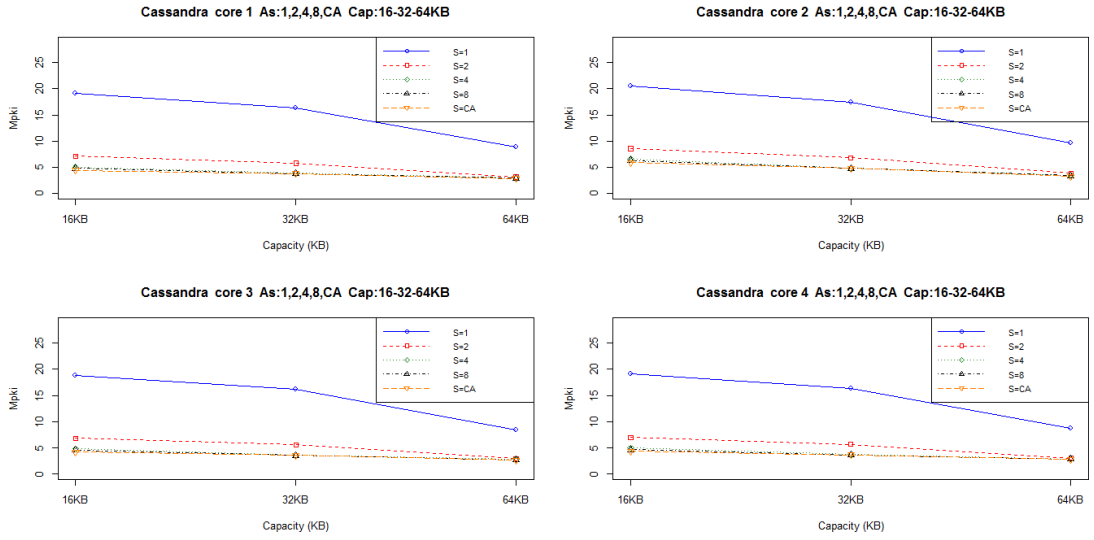

FIGURA 5.2: Cassandra: mpki de la cache de instrucciones en cada core vs. tamaño y asociatividad. Tamaño de bloque 64B.

5.1.3. Nutch (Figura 5.3)

Buen aprovechamiento de la capacidad y de la asociatividad: al aumentar tamaño de 16KB a 64 KB, nos movemos desde la franja 20-10 mpki a 5-1 mpki, para asociatividades entre 1 y CA, respectivamente. Todos los cores parecen ejecutar el mismo código.

5.1.4. Classification (Figura 5.4)

Podemos apreciar una gran diferencia entre asociatividad 1 y el resto. Independientemente del tamaño,a partir de asociatividad 4-8, la tasa de fallos es inapreciable.Todos los cores parecen ejecutar el mismo código.

5.1.5. Cloudstone (Figura 5.5)

El core 2 presenta una tasa de fallos (20-15 mpki) superior al resto (15-20 mpki), que se comportan de forma similar. Independientemente de la asociatividad, todos los cores experimentan el mismo descenso de mpki al doblar el tamaño, un 21 % aproximadamente. A partir de asociatividad 4 apenas se aprecia mejora.

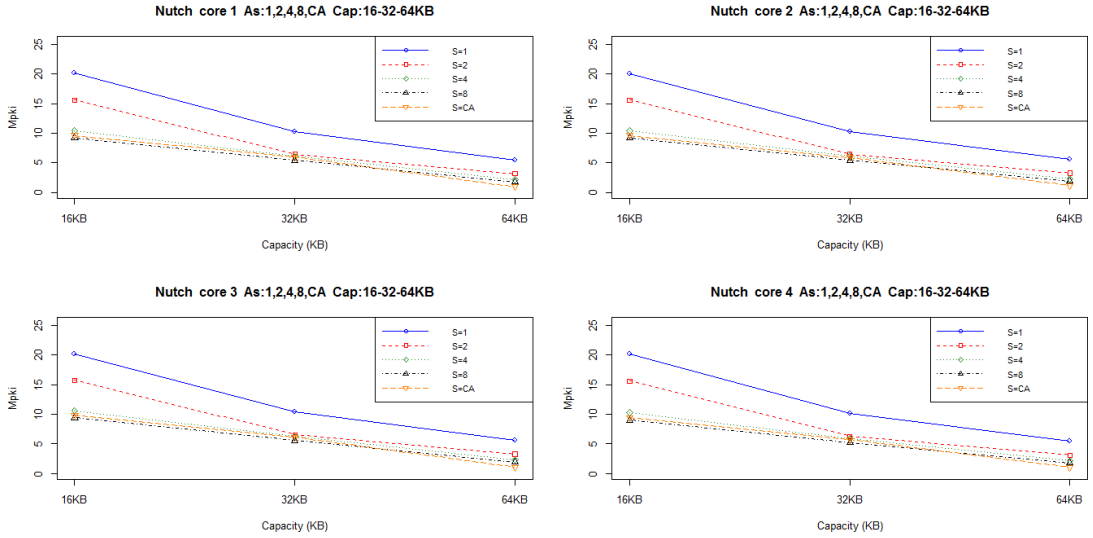

FIGURA 5.3: Nutch: mpki de la cache de instrucciones en cada core vs. tamaño y asociatividad. Tamaño de bloque 64B.

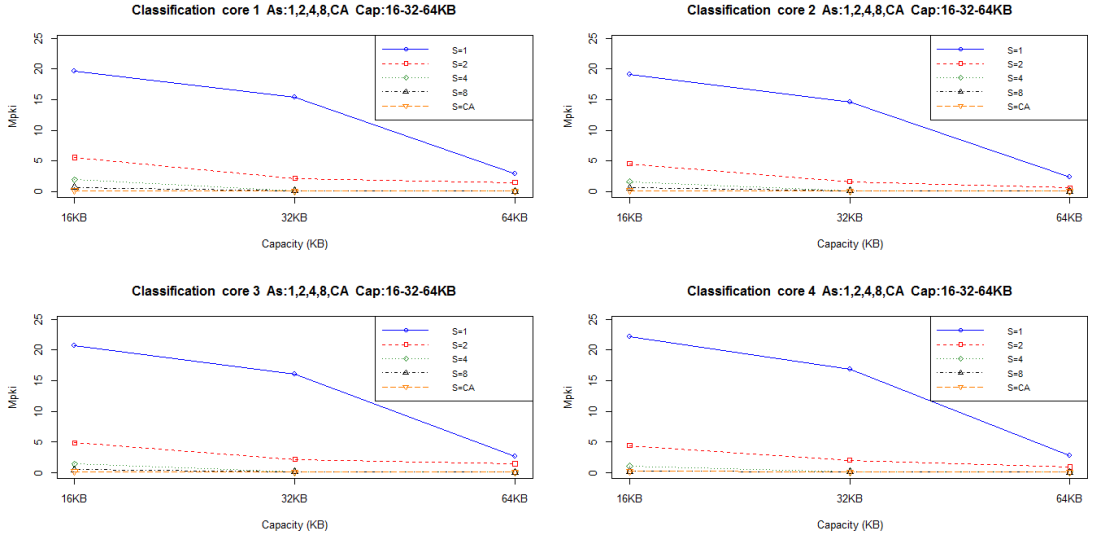

FIGURA 5.4: Classification: mpki de la cache de instrucciones en cada core vs. tamaño y asociatividad. Tamaño de bloque 64B.

5.1.6. Conclusiones

En base al estudio de las tasas medias de fallos por core, podemos extraer la siguientes conclusiones para el conjunto de todas las aplicaciones:

- Dos aplicaciones, Streaming y Cloudstone, no facilitan un diseño homogéneo de la cache de instrucciones, ya que un core se desmarca del comportamiento de los otros tres. Escoger descuidadamente una configuración de tamaño y asociatividad puede resultar en unas tasas de fallos excesivas para unos y en un diseño sobredimensionado para otros. La existencia de anomalías de asociatividad complica aún mas la decisión de diseño.
- El rango de tasa de fallos observado es grande, destacando Streaming, que puede llegar a los 50 mpki. Le sigue Cloudstone, presentando entre 20 y 10 mpki. A continuación, Nutch puede llegar a fallar bastante con pequeños tamaños (20-10 mpki), pero con suficiente capacidad y asociatividad apenas falla (5-1 mpki). Finalmente, Cassandra y Classification, con una asociatividad suficiente, apenas fallan (<4 mpki).
- A la vista de los experimentos realizados podemos derivar algunas pautas de diseño para la cache de instrucciones de primer nivel con tamaño de bloque 64 Bytes: - Si el tiempo y la energía de acceso no quedan comprometidas[2], el diseño mas razonable es un tamaño de 64 KB con asociatividad 4-8.
  - La opción de 32 KB es mas barata y rápida. Salvo para Streaming sería muy apropiada. Una asociatividad 4 sería suficiente.
  - En caso de optar por 16 KB, la mitad de las aplicaciones funcionarían bien por debajo de su potencial (Streaming, Nutch y Cloudstone). En esta caso, una asociatividad 4 también sería suficiente.

5.2. Huella de Memoria

La huella de memoria es el número total de bloques diferentes que un programa visita cuando se ejecuta. En nuestro caso nos interesa la huella de instrucciones, medida con una granularidad de 64 bytes, el tamaño de bloque de cache que vamos a utilizar en

[2] Tanto el tiempo como la energía de un acceso de cache crecen más o menos linealmente con el tamaño y de forma marginal con la asociatividad. Un diseño comercial no sólo considera la tasa de fallos, sino el tiempo medio de acceso y el posible impacto sobre el tiempo de acceso del procesador [HP06]

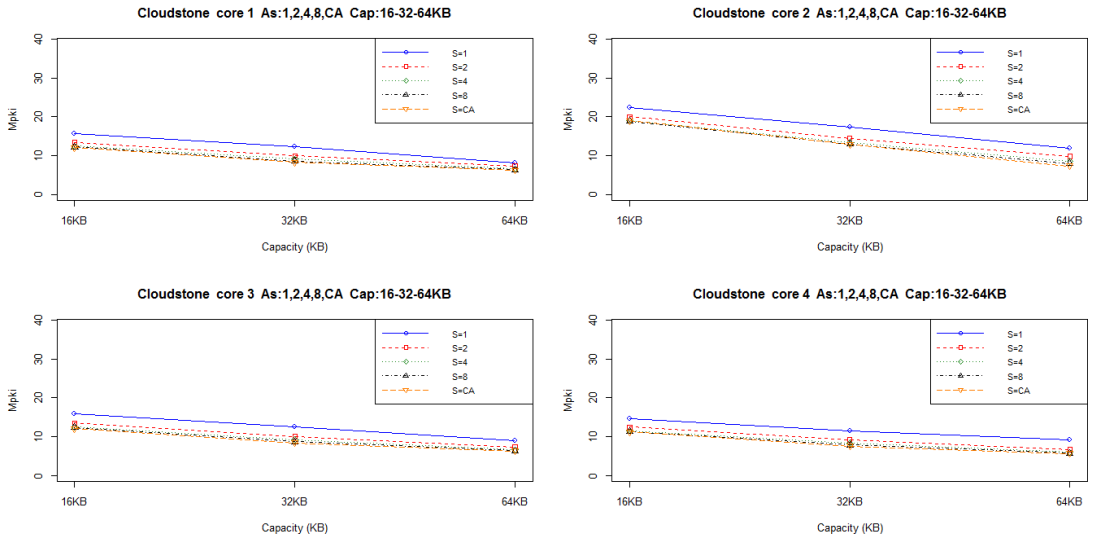

FIGURA 5.5: Cloudstone: mpki de la cache de instrucciones en cada core vs. tamaño y asociatividad. Tamaño de bloque 64B.

todo este capítulo. Por tanto la huella de instrucciones es equivalente al tamaño *efectivo* del código que se ha ejecutado en cada aplicación.

Para observar la evolución temporal, hemos medido en primer lugar la *huella de recarga* en intervalos de 100 ms. Para cada core, la huella de recarga mide el número de bloques diferentes que se visitan en cada intervalo. Esta medida se ha realizado mediante una cache completamente asociativa lo suficientemente grande para que no haya fallos de conflicto ni de capacidad, solo fallos obligatorios, que son los correspondientes al número de bloques diferentes referenciados. Al principio de cada intervalo se vacía la cache. Puesto que cada cache tiene su propia dinámica, hemos optado por representar únicamente la mayor de las cuatro huellas de recarga[3].

En las figuras también se presentan los resultados para cada core de la *huella acumulada*, que se ha calculado sin perder memoria cada 100 ms. Por tanto la huella acumulada al final de los 10 s representa el número total de bloques de instrucciones visitado por cada core.

Hay que prestar especial atención en las figuras ya que no todas están escaladas igual y no todos los ejes verticales comienzan en el valor cero.

En este apartado nos interesa verificar si las aplicaciones están en régimen estacionario, como afirman los creadores de los checkpoints. En tal caso, podríamos buscar fases de ejecución que nos permitan simular en una ventana de tiempo mas reducida. Por otra

[3]En el capítulo de conclusiones y líneas abiertas (Capítulo 6) se comenta un interesante trabajo futuro relacionado con las posibles similitudes o diferencias entre las huellas de recarga de los cuatro procesadores.

parte, también nos interesa descubrir si existe relación, o no, entre la huellas (acumulada o de recarga) y las tasas de fallos presentadas en el apartado anterior.

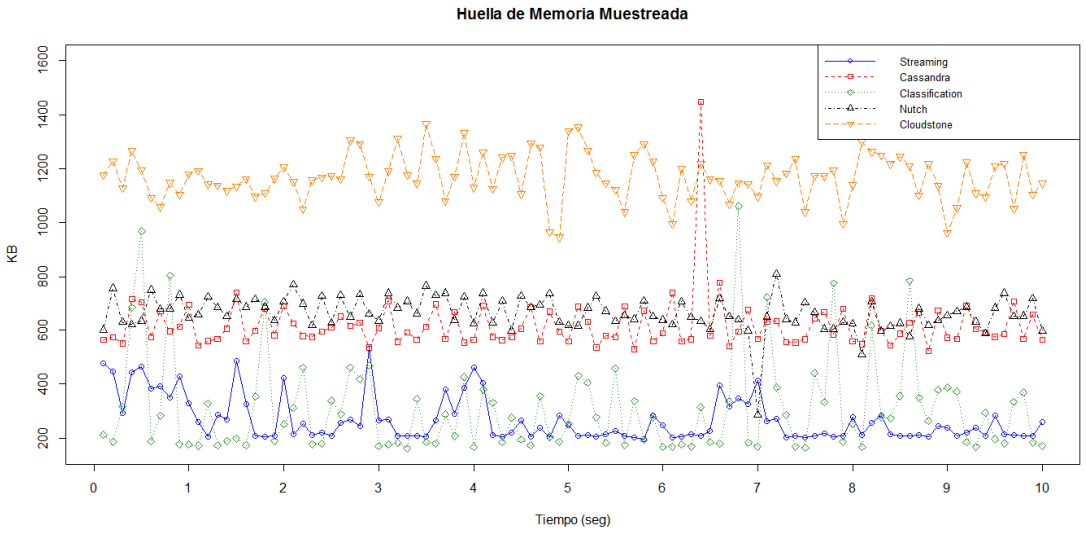

FIGURA 5.6: Huella de recarga de todas las aplicaciones de Cloudsuite. Cien intervalos de 100 ms. Tamaño en KB (2^{10})

En la figura 5.6 podemos ver la huella de recarga en KB (2^{10} bytes) para cada una de las cinco aplicaciones. Podemos observar que Cloudstone es la que más código toca, al contrario que Streaming y Classification que presentan huellas incluso 7 veces menores. Sin embargo, estas dos aplicaciones, sobre todo Classification, presentan una variabilidad relativa muy grande en su recarga. Nutch y Cassandra tocan una cantidad de código parecida, del orden de los 600-700 KB, mostrando bastante estabilidad en la cantidad de código que se recarga.

A continuación presentamos los resultados desagregados por aplicación.

5.2.1. Streaming (Figura 5.7)

La huella acumulada en los 10 segundos de simulación se mueve en un rango de entre 800KB para el core 3 y casi 600KB para el core 4. La recarga en cada muestra varía entre 200 KB y más de 500 KB.

Al principio de la ejecución los 4 cores cargan el código y después se pueden destacar los cambios de fase del core 2 y 3, que cargan unos 100 KB de código sobre el segundo 1,5 y el segundo 3, respectivamente. Estos cambios de fase corresponden a picos máximos en la huella de recarga.

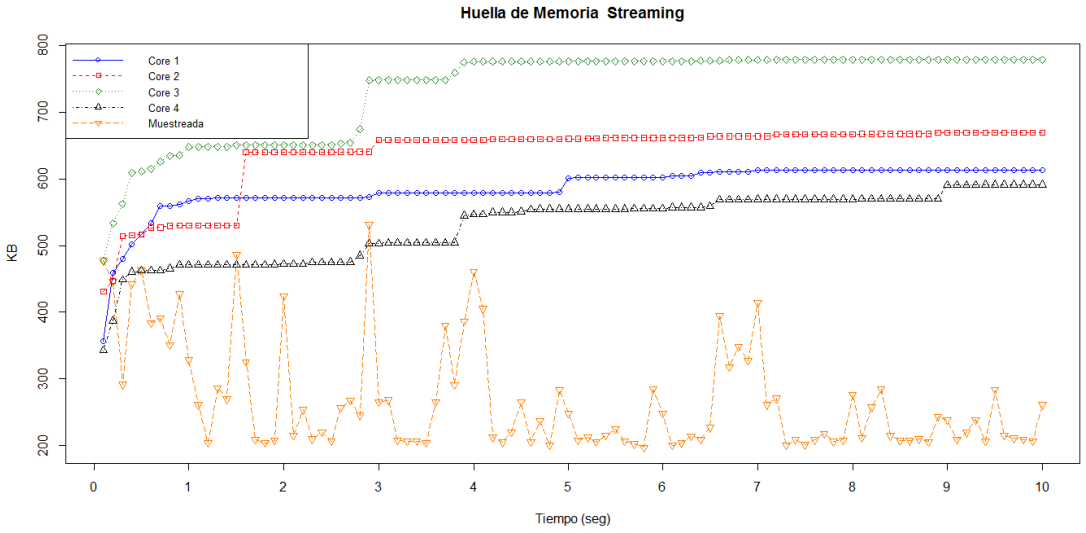

FIGURA 5.7: Huellas acumuladas y de recarga de de Streaming.

5.2.2. Cassandra (Figura 5.8)

Mientras que los cores 1,2 y 4 consumen entre 900 KB y 1200 KB de instrucciones a lo largo de los 10 segundos, el core 3 alcanza los 2000 KB. Hay varios cambios de fase en los cores, por ejemplo cerca del segundo 6 aparece una carga de unos 100KB de instrucciones en el core 1. Sin embargo, el que más destaca es el del core 3 en el segundo 6,4, que es de unos 900 KB. Esto coincide con el pico en la huella de recarga, que es unos 900 KB mayor que la media, media que se mantiene constante a lo largo del tiempo en unos valores de 500-700KB.

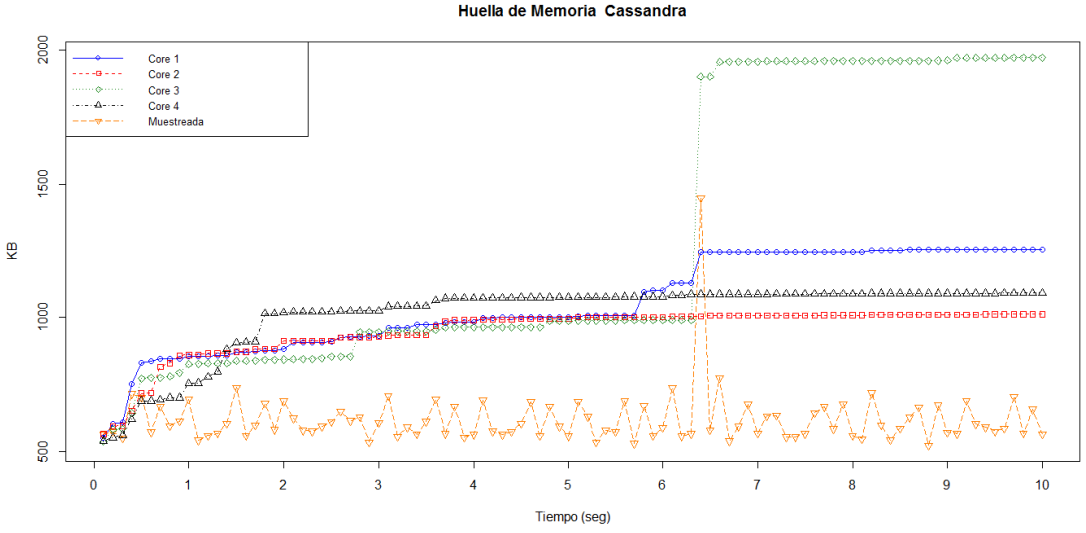

FIGURA 5.8: Huellas acumuladas y de recarga de Cassandra.

5.2.3. Nutch (Figura 5.9)

El número de bloques referenciados en total en los 10 segundos de simulación se mueve en un rango de entre 1200KB para el core 4 y 1000 KB para el core 3. Y la huella de recarga oscila entre 600 KB y 800 KB, destacando un punto muy bajo de unos 200KB.

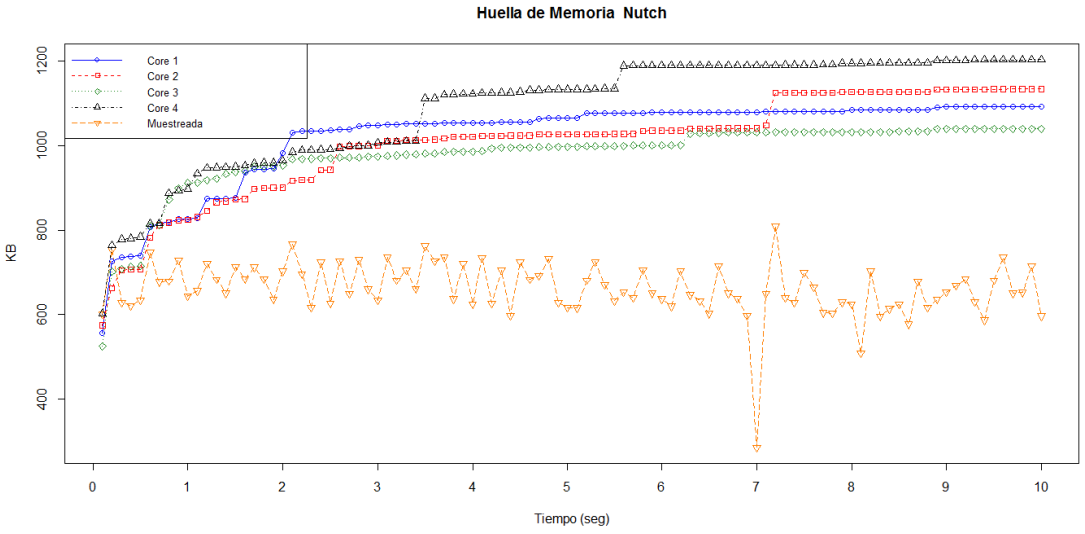

FIGURA 5.9: Huellas acumuladas y de recarga de Nutch.

5.2.4. Classification (Figura 5.10)

La huella acumulada en los 10 segundos de simulación se mueve en un rango de entre 2500KB para el core 1 y 1800KB para el core 2. Aunque las huellas de recarga son relativamente bajas, moviéndose en su mayoría por debajo de los 500KB, existen picos significativos que corresponden con los cambios de fase de las huellas acumuladas.

5.2.5. Cloudstone (Figura 5.11)

Alcanzamos una huella acumulada superior a los 4000KB, mucho más alto que en el resto de las aplicaciones. Observamos que los 4 cores tienen un comportamiento prácticamente idéntico, trazando una función de aspecto logarítmico. Al principio se cargan grandes tamaños, del orden de 100KB, y al final casi no se cargan nuevos bloques de instrucciones. No hay cambios de fase claros ya que la carga es progresiva, y esto se puede apreciar en la huella de recarga, ya que varia poco y no tienen ningún pico significativo.

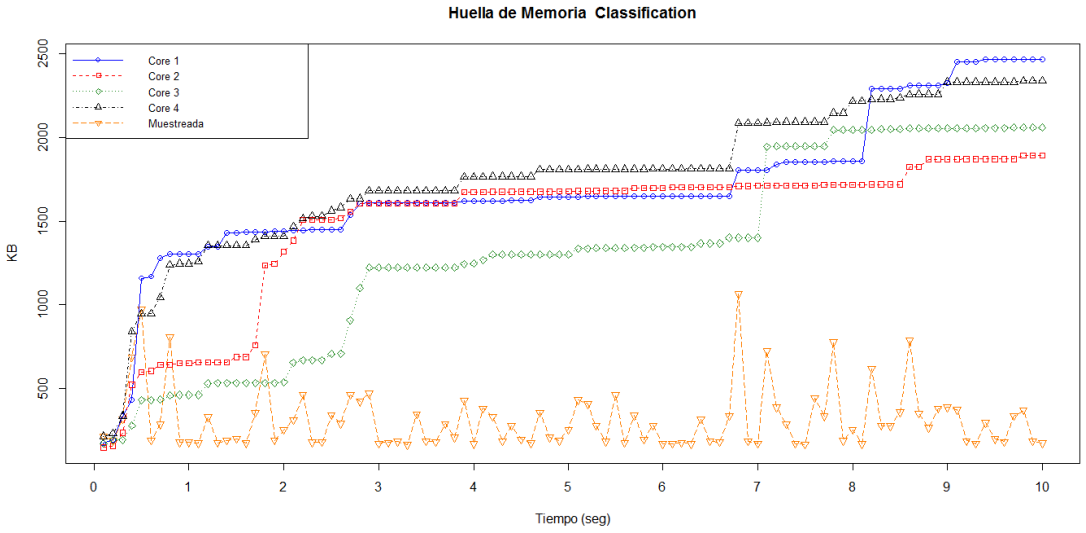

FIGURA 5.10: Huellas acumuladas y de recarga de Classification.

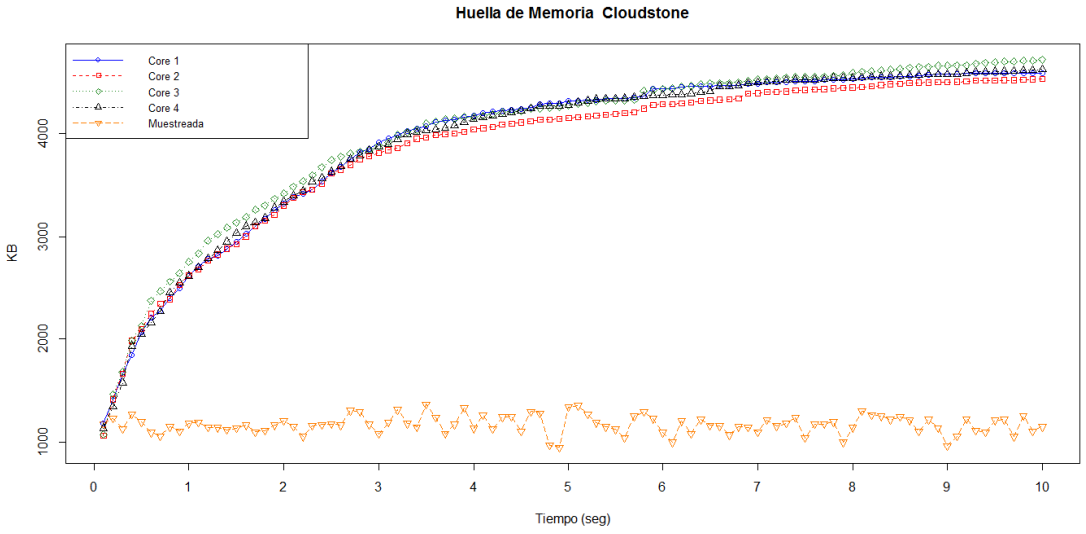

FIGURA 5.11: Huellas acumuladas y de recarga de Cloudstone.

5.2.6. Conclusiones

- Los creadores de los checkpoints afirman que el punto de inicio de ejecución corresponde a un régimen estacionario. Sin embargo todas las aplicaciones no estabilizan sus huellas hasta bien entrada la ejecución. En concreto, a partir del segundo 4 para Streaming, 6,4 para Cassandra, 7,2 para Nutch, 7 para Cloudstone y 9 para Classification. Sin embargo sería necesario simular unos 10 segundos más para ver si realmente se ha llegado a un comportamiento estable en cuanto a huella acumulada y recarga.

- Cuando hablamos de un cambio de fase, nos referimos a que recargamos del orden de 100 o más KB de instrucciones. Estos cambios de fase, que se dan en todas las aplicaciones excepto en Cloudstone, complican escoger una ventana de tiempo representativa.

- No existe correlación entre las huellas de memoria y las tasas de fallos observadas en el apartado anterior. Por ejemplo, Streaming no destaca ni por el tamaño final de la huella ni por sus valores de recarga, pero es la aplicación que mas falla: se visita poco código, pero con poca localidad. Por el contrario, a veces se visita mucho código, pero con gran localidad. En este grupo caen Cassandra y Cloudstone. Cassandra falla realmente poco, aunque su huellas acumuladas y de recarga son de las mayores de la suite. Cloudstone es con diferencia la que tiene huellas acumuladas y de recarga mas grandes. Sin embargo, sus tasas de fallos son medias.

5.3. Ancho de Banda de instrucciones

Como ya hemos indicado en el apartado 4.1, la métrica *BWin* es el ancho de banda de instrucciones que entra desde el siguiente nivel, agregado para las cuatro caches. La figura 5.12 recuerda el sistema simulado y resalta la agregación de los tráficos entrantes a las cuatro caches de instrucciones.

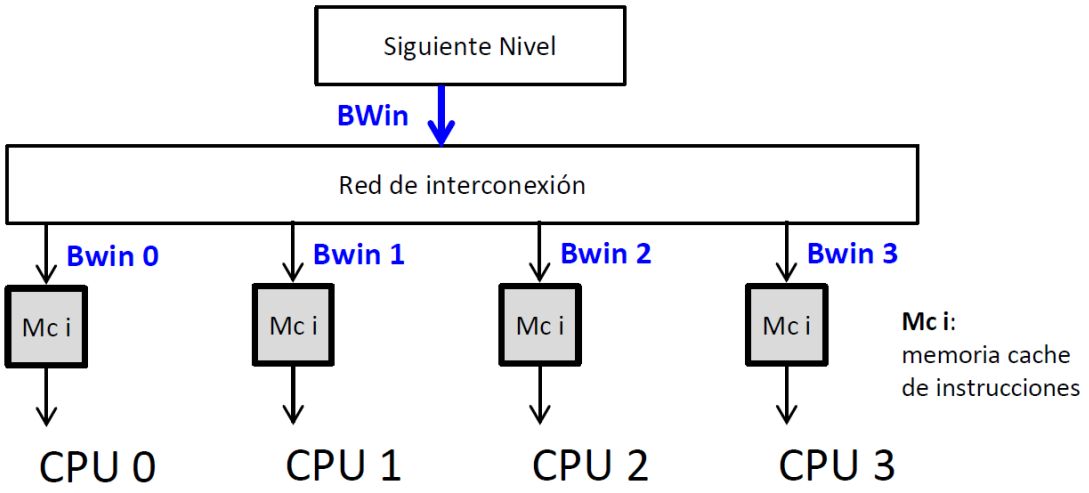

FIGURA 5.12: Sistema simulado para calcular Bwin, el ancho de banda de instrucciones.

La siguiente formula concreta como se calcula dicho ancho de banda a partir de las tasas de fallos en mpki, asumiendo un procesador de 2 GHz que ejecuta a un ritmo de un ciclo

por instrucción:

$$BWin(GBps) = \sum_{K=1}^{4} mpki_{core\ K} \times \frac{64B}{1000\ instrucciones \times 1\frac{ciclo}{instrucción} \times 0,5\frac{ns}{ciclo}} = \sum_{K=1}^{4} mpki_{core\ K} \times 0,128 \times \frac{10^9}{2^{30}} GBps \tag{5.1}$$

Vamos a presentar en tres gráficas la evolución temporal de BWin para cada tamaño de cache (16KB, 32KB, 64KB); en cada gráfica se detalla el comportamiento para cada asociatividad.

5.3.1. Streaming (Figura 5.13)

En concordancia con sus elevadas tasas de fallos, Streaming es la aplicación que puede presionar más al siguiente nivel, presentando picos de hasta 25 GBps para la cache mas sencilla (16 KB, S=1). Al igual que en todo el resto de aplicaciones, si consideramos la evolución temporal como una señal, podemos ver como el aumento de tamaño y asociatividad actúa como un filtro paso bajo con gran reducción de la componente continua. En este caso, al pasar de 16 KB con S=1 a 64 KB con S=2, BWin se reduce, en media, en un orden de magnitud.

Al aumentar el tamaño de cache, el BWin es más sensible a la asociatividad: para 16 KB apenas se nota la diferencia entre asociatividades, mientras que para 64 KB cada vez que doblamos la asociatividad, el BWin desciende apreciablemente. Hay que resaltar que el resto de aplicaciones van a presentar justo el comportamiento contrario: a mayor tamaño, menor impacto de la asociatividad.

5.3.2. Cassandra (Figura 5.14)

Hay que destacar la gran diferencia entre correspondencia directa y el resto de asociatividades, aunque ésta disminuye bastante al aumentar el tamaño de la cache. También observamos que a partir de una asociatividad mínima (S >2), apenas hay diferencias de ancho de banda.

La influencia del tamaño de la cache sobre el filtrado de ruido y la disminución de la componente continua sigue el patrón general.

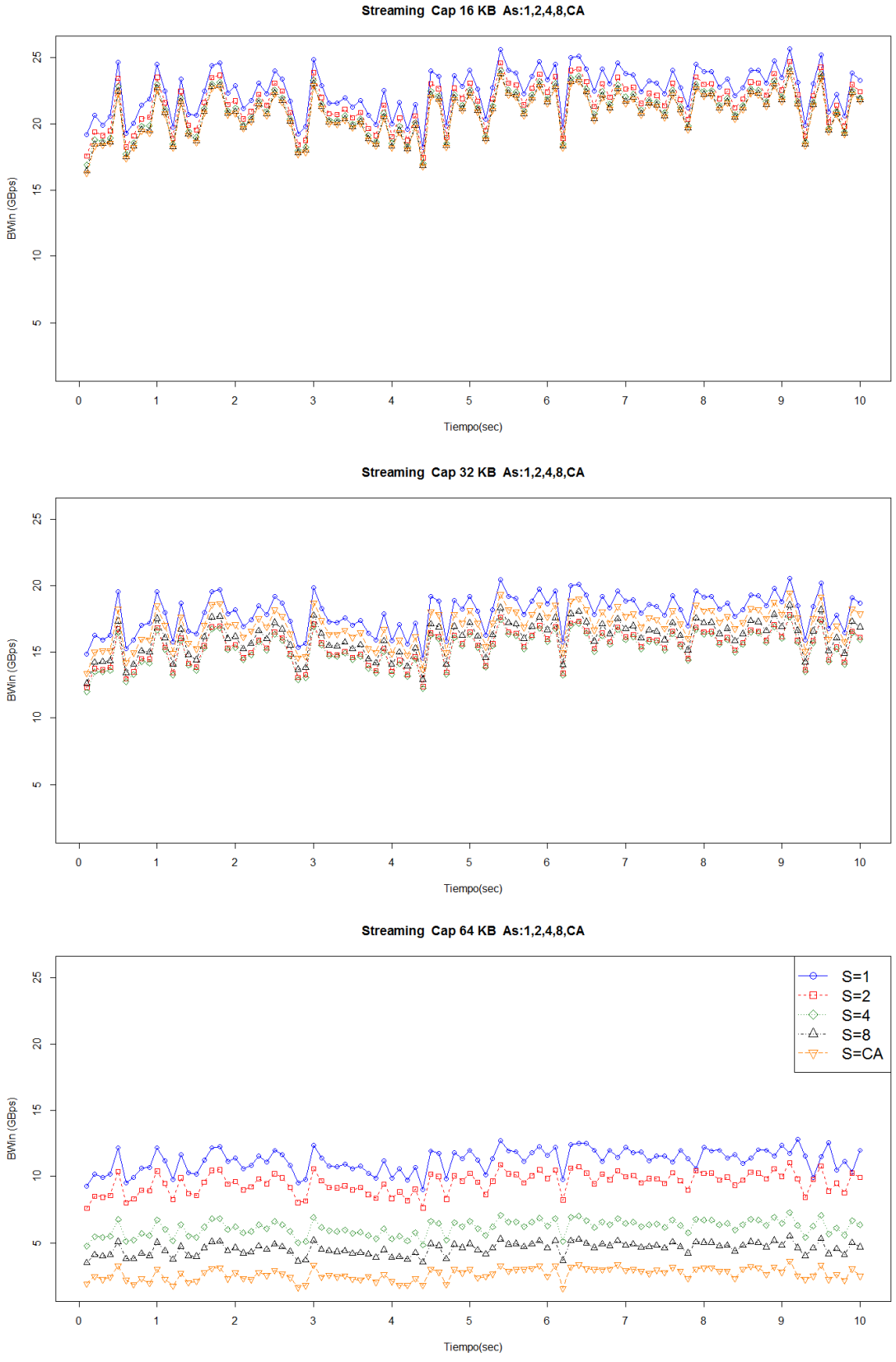

FIGURA 5.13: Ancho de banda por asociatividad y tamaño en Streaming.

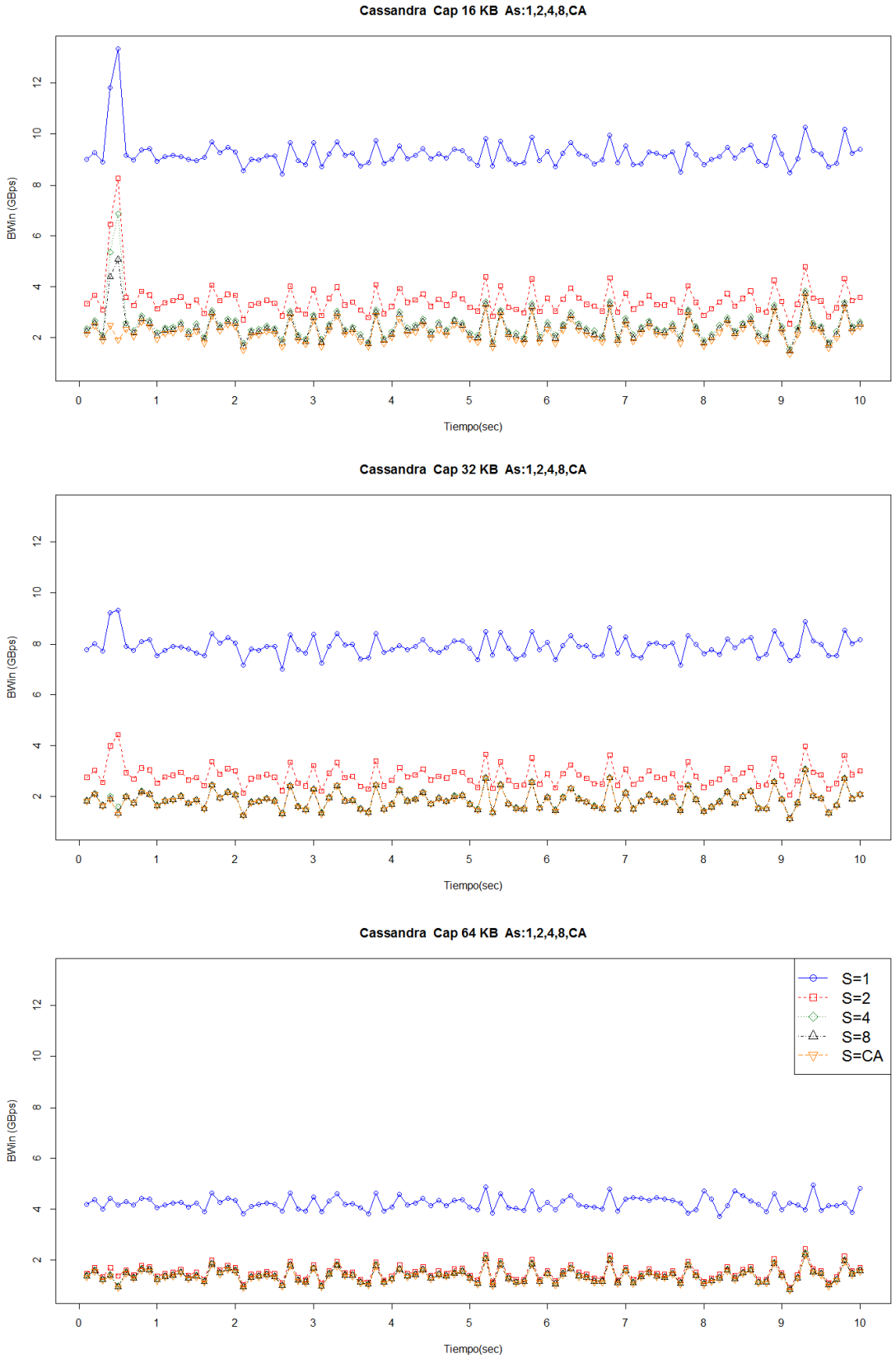

FIGURA 5.14: Ancho de banda por asociatividad y tamaño en Cassandra.

5.3.3. Nutch (Figura 5.15)

Destaca la gran variabilidad temporal del ancho de banda, sobre todo para 16KB y 32KB. Al aumentar el tamaño de la cache el rango del ancho de banda se reduce apreciablemente, aunque el escalado es variable (e.g. en media, el cociente entre BWin para S=1 y para CA, es del orden de 2, 1,7 y 6, para las caches de 16, 32 y 64 KB, respectivamente).

La influencia del tamaño de la cache sobre el filtrado de ruido y la disminución de la componente continua sigue el patrón general.

5.3.4. Classification (Figura 5.16)

Salvo con una cache de correspondencia directa, esta aplicación es la que menos presiona al siguiente nivel de memoria. Observamos una gran diferencia entre asociatividad 1 y el resto de asociatividades. Además, al aumentar el tamaño de la cache, la sensibilidad a la asociatividad es menor.

La influencia del tamaño de la cache sobre el filtrado de ruido y la disminución de la componente continua sigue el patrón general.

5.3.5. Cloudstone (Figura 5.17)

Junto con la aplicación Nutch, Cloudstone destaca por gran variabilidad temporal. Pero en este caso, apenas es apreciable el filtrado paso bajo que se observa en el resto de aplicaciones al aumentar el tamaño de cache. Observamos que la diferencia absoluta entre las distintas asociatividades es casi constante, independientemente del tamaño de la cache.

5.3.6. Conclusiones

- Con tan solo cuatro procesadores las aplicaciones estudiadas pueden ejercer una presión notable sobre el siguiente nivel de memoria cache. Si asumimos que la recarga de instrucciones se produce regularmente, sin ráfagas (lo cual no suele ser cierto), podemos calcular a partir de BWin el número medio de ciclos entre las transacciones de 64 B, mediante la siguiente formula (procesadores de 2GHz):

$$\frac{119,2}{BWin(GBps)} \cdot \frac{ciclos}{transaccion} \tag{5.2}$$

  Esto significa que un ancho de banda de 10 GBps, observado en mas de una aplicación y configuración, supone en media un acceso cada 12 ciclos, aproximadamente.

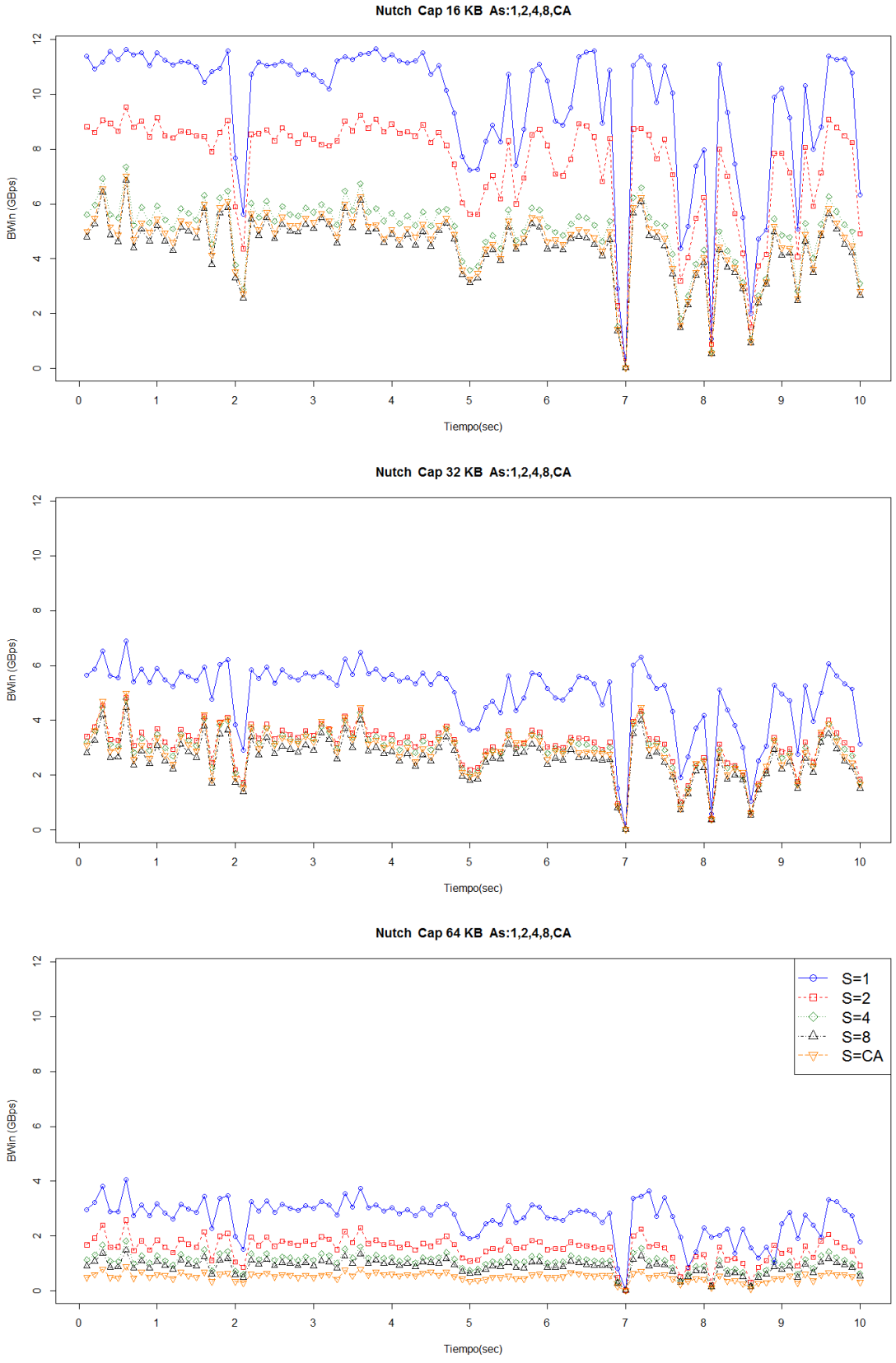

FIGURA 5.15: Ancho de banda por asociatividad y tamaño en Nutch.

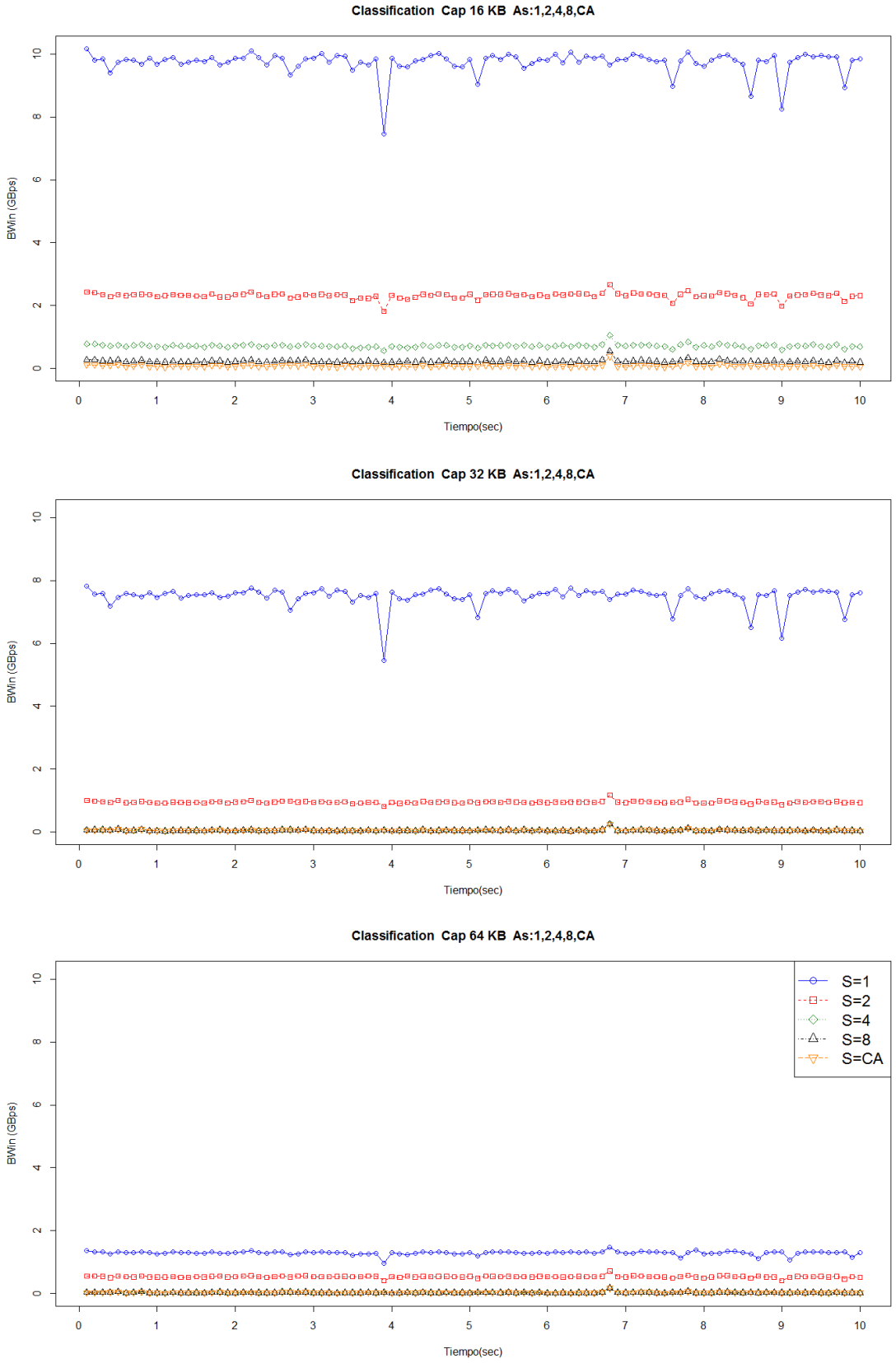

FIGURA 5.16: Ancho de banda por asociatividad y tamaño en Classification.

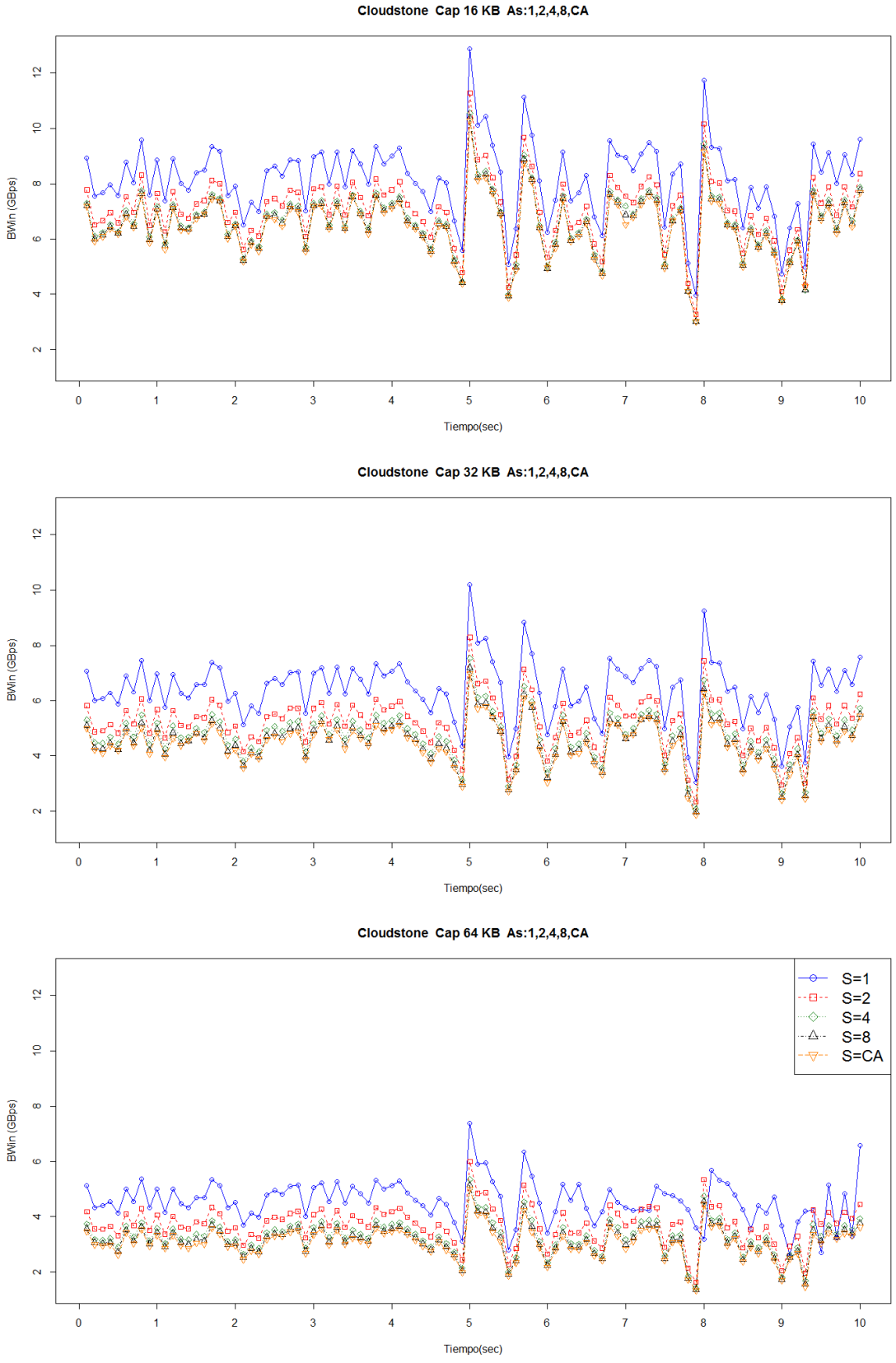

FIGURA 5.17: Ancho de banda por asociatividad y tamaño en Cloudstone.

En la actualidad, las latencias de las caches de segundo nivel están en ese orden de magnitud, lo cual hace pensar en la necesidad de un diseño específico. Una posibilidad sería un siguiente nivel de memoria cache de instrucciones *privado* para cada core, que disminuiría el tráfico por cuatro, ver figura 5.18(a). Otra posibilidad sería un siguiente nivel multibanco para datos e instrucciones, que permita el servicio simultáneo del tráfico de datos (que no ha sido considerado en este trabajo) y soporte la presencia de ráfagas de fallos, cuya serialización también comprometería las prestaciones, ver figura 5.18(b).

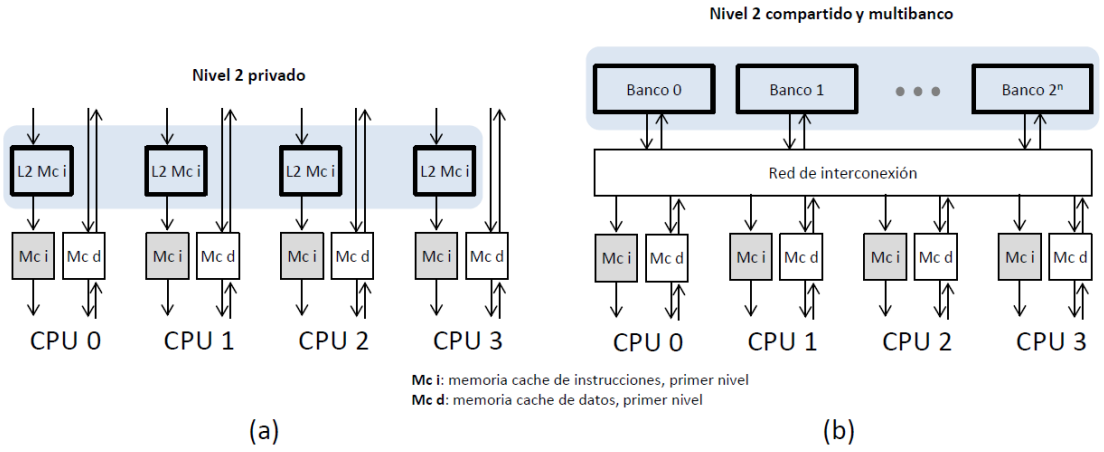

FIGURA 5.18: Dos propuestas para mejorar el suministro de instrucciones desde el siguiente nivel.(a)Segundo nivel de cache de instrucciones privado para cada core.(b)Segundo nivel de cache compartido (datos + instrucciones) y multibanco.

- Una buena forma de entender como afecta el aumento de tamaño y asociatividad consiste en observar la evolución temporal del ancho de banda como si fuera una señal, y considerar a la cache como un filtro paso bajo, que de acuerdo a su tamaño y asociatividad, reduce progresivamente el nivel de la componente continua y su frecuencia de corte. Esta es una hipótesis interesante, que debería ser contrastada de forma rigurosa, pero que excede del alcance de este trabajo.

- Desde el punto de vista de muestreo, es decir, de la posibilidad de extraer conclusiones válidas observando únicamente una parte pequeña de toda la evolución temporal, podemos dividir las aplicaciones en dos grupos. El primero está formado por Nutch y Cloudstone; su variabilidad es grande y no parece que pueda escogerse un trozo pequeño representativo. El segundo está formado por Streaming, Cassandra y Classification; en estos casos, no parecen verse fases claras, pareciendo cualquier tramo similar y susceptible de ser representativo. Por supuesto estas reflexiones valen únicamente para experimentos encaminados a probar el siguiente nivel, cargándolo con un tráfico representativo.

5.4. Comparación con otras cargas de trabajo

Según los creadores de Cloudsuite una de sus principales características es que sus benchmarks ejercen una fuerte presión contra la cache de instrucciones [FAK+12]. Para contrastar esta afirmación vamos a comparar las tasas de fallos que hemos obtenido experimentalmente con los suyos y con otras fuentes disponibles. A continuación presentamos un resumen de los trabajos que hemos consultado:

- ***Evaluating associativity in CPU caches***. Este es el trabajo en el que Hill y Smith proponen el modelo de las 3Cs [HS89]. Además, en él realizan un estudio para arquitecturas maduras de 32 bits, mediante simulaciones hechas con 28 trazas de computadores IBM 370 (sistema operativo MVS) y DEC VAX-11 (sistemas operativos VMS y ULTRIX). Los datos que reproducimos corresponden a unos promedios ajustados que sus autores denominan "design target miss ratios", pensados para caracterizar de forma tabular el comportamiento "medio"de una cache y no tener que simular. En la figura 5.19 están recogidos los mpki para caches de instrucciones de tamaño 16KB y 32KB y asociatividades desde 1 a 8, con tamaño de bloque 64B.

- ***Filtering Directory Lookups in CMPs***. En esta tesis Ana Bosque recoge la tasa de fallos para las aplicaciones de la suite SPLASH2, una suite usada para estudios científicos de máquinas paralelas con memoria compartida [LVIB11]. Las simulaciones se realizaron con SIMICS, con binarios compilados para SPARC v9 en Solaris 8, y corresponden a una cache de instrucciones de primer nivel de 16 KB, con tamaño de bloque 32B y asociatividad 8. En la figura 5.19 se reproducen las tasas de fallos obtenidas para toda la sección paralela de cada benchmark.

- ***Memory System Behavior of Java-Based Middleware***. En este trabajo, Karlsson, Moore, Hagersten y Wood usan dos benchmarcks: SPECjbb y ECperf [KMHW03]. Las simulaciones se realizaron con SIMICS, con binarios compilados para SPARC v9 en Solaris 8. *SPECjbb* está diseñado para medir la habilidad de un sistema para ejecutar aplicaciones Java en el lado del servidor. Esta aplicación conecta a los clientes con la base de datos a través de la lógica de negocio, pero para hacer el benchmark más portable y fácil de usar, no usan una base de datos comercial, almacenando directamente las tablas en memoria como objetos de tipo árbol de Java. *ECperf* está diseñado para comprobar el rendimiento y la escalabilidad de un sistema de 3 niveles (cliente, servidor y base de datos), modelando un negocio online. Los benchmarks se simulan para distintos tamaños de cache, con asociatividad 4 y tamaño de bloque 64B.

- ***Clearing the clouds***. Este es el trabajo que motiva en parte nuestro estudio. En él, Ferdman et al. del laboratorio Parsa realizan un estudio de la Cloudsuite en un hardware real, bajo sistema operativo Linux y utilizando contadores de prestaciones. La máquina que aloja los benchmarks a monitorizar es un Dell PowerEdge M1000e, con dos procesadores Intel X5670 y 24GB de RAM en cada blade. Cada procesador Intel X5670 incluye seis cores agresivos con ejecución fuera de orden, con una jerarquía de tres niveles de cache. El primer nivel es privado y separado para datos e instrucciones. El segundo también es privado, pero contiene datos e instrucciones. Finalmente, el tercer y último nivel es compartido por los seis cores. La cache L1 de instrucciones real tiene tamaño 32KB, asociatividad 4 y tamaño de bloque 64B. Sus tasas de fallos recogen 180 segundos por cada carga de trabajo, una vez que se ha completado la fase de inicialización de carga y se supone que el sistema ha entrado en un régimen estacionario[FAK+12]. Se supone que las tasas de fallos que reproducimos promedian el comportamiento de los 12 procesadores, aunque esto no está explicitado en su artículo.
- ***Nuestro trabajo***. Para comparar con un número único se ha calculado la mediana en cada benchmark de todas las muestras temporales para los cuatro procesadores.

La información derivada de estos trabajos se ha resumido en la tabla de la figura 5.19. En la columna de la izquierda está la fuente. En la segunda columna presentamos las aplicaciones. En las siguientes columnas aparecen las tasas de fallos de instrucciones para diferentes tamaños y asociatividades, siempre que ha sido posible para tamaño de bloque 64 B. En la última columna se hace referencia a la naturaleza de las instrucciones consideradas, ya sea de sólo de usuario (u), o de sistema y usuario a la vez (u+s y u∪s). En la suite SPLASH2 únicamente se considera actividad de usuario, pero al tratarse de aplicaciones científicas, es conocido que suponen una carga de sistema despreciable. En el trabajo de referencia de Cloudsuite se desagrega la actividad de usuario y de sistema (u+s) [FAK+12], mientras que en el resto de trabajos no se dispone de ese detalle (u∪s).

5.4.1. Conclusiones

Los datos recogidos son heterogéneos y escasos, pero comparando nuestras tasas con las del resto de los autores, podemos extraer algunas conclusiones:

- **Trabajo de Hill y Smith [HS89].** Si comparamos la media de nuestras aplicaciones con sus números, ambos resultados son muy similares. Si descartamos a Streaming y Classification por atípicos (*outliers*), entonces nuestros resultados suponen tasas de fallos menores.

MPKI de instrucciones

	Tamaño	16KB		32KB	64KB	Tipo de Actividad
	Asociatividad	4	8	4	4	
[HS89]	Trazas	19	18	11	-	u ∪ s
SPLASH2 1995 Bloques de 32 B	Barnes	-	4	-	-	u
	Fmm	-	16	-	-	
	Ocean	-	8	-	-	
	Radiosity	-	177	-	-	
	Raytrace	-	8	-	-	
	Volrend	-	30	-	-	
	Water-nsquared	-	1	-	-	
	Water-spatial	-	1	-	-	
[KMHW03]	SPECjbb	12	-	-	6	u ∪ s
	ECperf	17	-	-	10,5	
[FAK+12]	Streaming	-	-	35+35	-	u + s
	Cassandra	-	-	24+21	-	
	CloudStone	-	-	20+22	-	
	Nutch	-	-	7+3	-	
Propios (Mediana)	Streaming	46,6	46,2	33,70	13,1	u ∪ s
	Cassandra	4,9	4,7	3,7	2,8	
	CloudStone	13,3	13,1	9,7	6,8	
	Nutch	11	9,5	6,10	2,2	
	Classification	1,4	0,2	0,1	0,1	

FIGURA 5.19: Comparación MPKI benchmarks. Tipo de actividad hace referencia a si los datos son de usuario (u) o de sistema (s)

- **Tesis de Ana Bosque [LVIB11].** La media de estas aplicaciones ronda los 30 mpki, superior a nuestra media, pero destaca la gran influencia del atípico Radiosity y el efecto, notable, de utilizar un tamaño de bloque de 32 B. Si descartamos los atípicos, la media de este trabajo queda por debajo de la nuestra, lo cual es razonable para cargas científicas.

- **Trabajo de Karlsson et al. [KMHW03].** Muy comparable a nuestros resultados. Podríamos colocarlos como propios y pasarían desapercibidos.

- **Trabajo de Ferdman et al. [FAK$^+$12].** Esta comparación tiene un gran interés, puesto que estamos hablando de las mismas aplicaciones. Sin embargo, en la confrontación uno a uno, tan solo la aplicación Nutch presenta tasas comparables. El resto de aplicaciones presenta tasas *significativamente mayores* en las ejecuciones reales del Parsa. ¿Cómo explicarlo?. No lo sabemos. Es cierto que hay dos factores diferenciales que juzgamos importantes, el tamaño de la muestra y el sistema operativo. Nosotros simulamos 10 s y ellos ejecutan 180 s. Nuestro sistema operativo

es Solaris y el suyo Linux. Pero estos elementos no deberían causar una distorsión tan grande. Además, en nuestro caso no se ha podido distinguir entre la actividad de usuario y la de sistema, lo cual hace más difícil el análisis.

- En resumen, la comparación con las tres primeras fuentes, con aplicaciones diferentes, parece reforzar la veracidad de nuestros resultados, mientras que el contraste directo con los creadores de Cloudsuite coloca nuestras simulaciones en unas tasas de fallos demasiado bajas. Está claro que es necesario realizar mas trabajo para llegar a una explicación satisfactoria.

Capítulo 6

Conclusiones y lineas abiertas

6.1. Conclusiones técnicas

El objetivo de este proyecto ha sido caracterizar el comportamiento de las instrucciones en la suite Cloudsuite 2.0, un conjunto de aplicaciones cliente/servidor del grupo de investigación Parsa de la EPFL en Suiza.

Para ello hemos usado la plataforma de simulación Simics,un simulador de sistema completo, trabajando con las cinco aplicaciones de la suite que están acompañadas de checkpoints públicos para su simulación en Simics. Además, se ha escrito un tutorial de Simics, acompañado de material práctico, para facilitar y agilizar la fase de formación de otros proyectos que también utilicen esta plataforma.

Para realizar los experimentos deseados se han programado dos módulos de Simics de jerarquía de memoria basados en el módulo g-cache que implementan dos algoritmos eficientes y específicos para registrar tasas de fallos y huellas de memoria. Un algoritmo obtiene resultados para múltiples caches en una sola simulación (Algoritmo MC) y el otro está especializado en caches completamente asociativas (Algoritmo CCA). Se han realizado 6 y 5 experimentos respectivamente por cada aplicación, con una duración de 10 s para cada aplicación, destacando los siguientes resultados experimentales:

- Hemos analizado el comportamiento de las caches de instrucciones en cuanto a su tasa de fallos expresada en mpki, en función de su tamaño y de su asociatividad. En la comparación de los valores obtenidos con diferentes fuentes bibliográficas destaca la discrepancia con los creadores de Cloudsuite, que podría atribuirse a la diferente longitud de la simulación, o al uso de sistemas operativos diferentes. Además, en base a las tasas de fallos obtenidas, se han sugerido configuraciones prácticas de tamaño y asocitividad para cada aplicación.

- Hemos obtenido la evolución temporal de la huella de memoria de instrucciones, es decir, el tamaño efectivo del código que se ha ejecutado en cada aplicación. Esto se ha hecho tanto para la huella acumulada como para la de recarga. A partir del estudio de las huellas concluimos que todas las aplicaciones no entran en régimen estacionario hasta transcurridos muchos segundos de la aplicación, y que aparecen bastantes fases, lo cual va a complicar la selección de ventanas de simulación, representativas y de corta duración.
- Finalmente hemos obtenido la evolución temporal de BWin, el ancho de banda de instrucciones agregado para las cuatro caches que entra desde el siguiente nivel, para cada tamaño de cache y asociatividad. A partir de esta métrica se ha constatado que la presión ejercida sobre el siguiente nivel puede ser realmente grande, y se han sugerido configuraciones de ese segundo nivel con capacidad para absorber las demandas del primero.

6.2. Lineas abiertas

A partir de este trabajo han aparecido unas líneas de continuación que no han podido ser abordadas, ya que excedían del alcance previsto y del tiempo disponible:

- Estudiar las posibles similitudes o diferencias entre las huellas de recarga de los cuatro procesadores a través de algún procedimiento estadístico riguroso. En el fondo se trata de conocer, de forma sencilla, si cada procesador está ejecutando el mismo código o no.
- Para el diseño de las caches no sólo es necesario tener en cuenta la asociatividad y el tamaño de la cache, si no también su tiempo de acceso (latencia) y su consumo energético. Para realizar un estudio más profundo se podrían utilizar herramientas como CACTI[1], que devuelve éstos parámetros físicos a partir de una configuración dada (tamaño, asociatividad, tamaño de bloque, etc.). Con esta información se puede reevaluar la utilidad de la cache en términos de tiempo efectivo de acceso, que es mas preciso que la tasa de fallos.
- Tal y como ya apuntábamos en el apartado 5.4, en nuestros resultados nos hemos encontrado con unas tasas de fallos significativamente menores que las ejecuciones reales que realiza Parsa con Cloudsuite. Una de las principales diferencias con su ejecución es que ellos simulan 180 segundos frente a los 10 segundos nuestros. Por ello se propone, para seguir contrastando estos datos, lanzar nuevos experimentos cuya ejecución sean 180 segundos de aplicación real.

[1] http://www.hpl.hp.com/research/cacti/

- En el apartado5.3 hemos observado que si analizamos las gráficas del ancho de banda como si fueran una señal parece que la cache puede verse como un filtro paso bajo, que reduce progresivamente el nivel de la componente continua y su frecuencia de corte, en función de su tamaño y asociatividad crecientes. Parece muy atractivo recurrir a la teoría de procesado de señal para formalizar esta hipótesis, realizando el análisis espectral correspondiente, y decidiendo si es necesario aumentar la frecuencia de muestreo.

6.3. Conclusiones personales

Este proyecto me ha permitido aplicar aptitudes y conocimientos adquiridos en la carrera pero también adquirir algunos totalmente nuevos referentes al mundo de la investigación.

He conocido nuevas herramientas y una nueva forma de trabajar. En investigación no sabes cuales van a ser los resultado inicialmente, así que aunque sigue siendo muy necesaria una buena planificación, es inevitable que algo no salga tal cual se había planeado. Aunque esto puede resultar muy frustrante, también puede tener resultados muy positivos. También he aprendido que en estos proyectos es indispensable la cooperación con otras personas que te ayuden a ver los problemas desde distintas perspectivas, ya que se existen muchas variables a tener en cuenta.

Además, ha sido muy gratificante aplicar lo estudiado en las asignaturas de arquitectura de computadores y ver su utilidad en el mundo real.

En cuanto a mi futuro profesional me ha dado conocer un nuevo camino a considerar, el de la investigación, antes prácticamente desconocido para mí.

Apéndice A

Carga y Desarrollo del Proyecto

Este apéndice contiene detalles acerca de la gestión del tiempo y el esfuerzo invertido durante el proyecto, así como algunos problemas encontrados a lo largo de su desarrollo.

A.1. Gestión del tiempo

Este proyecto se ha desarrollado desde finales de septiembre de 2013 hasta agosto de 2014, en dedicación a tiempo parcial. En el diagrama de Gantt que se presenta en la figura A.1 se puede ver cómo se han distribuido las diferentes tareas a lo largo del tiempo.

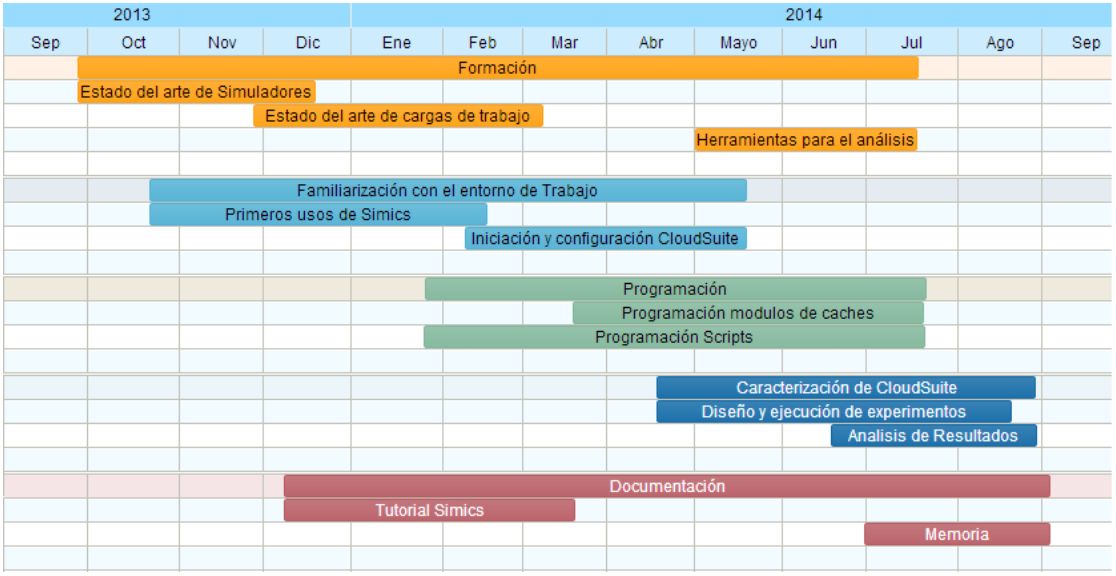

FIGURA A.1: Diagrama de Gantt del proyecto.

A continuación incluimos un pequeño resumen del trabajo que engloba cada tarea:

- **Formación.** La formación es una de las partes más importantes del proyecto, que se extiende en prácticamente su totalidad. Tanto las herramientas, cómo la forma de trabajar ya que es un proyecto de investigación, eran desconocidas y requirieron una gran cantidad de esfuerzo. En esta tarea está incluido el estudio del Estado del arte de Simuladores, de cargas de trabajo y de las herramientas necesarias para el análisis. En estas herramientas está incluido el aprendizaje del uso del programa R para producir las gráficas necesarias que nos servirán para analizar los resultados.
- **Familiarización con el entorno de trabajo.** Esta tarea consiste en el aprendizaje del simulador SIMICS, y la configuración y puesta a punto de la Cloudsuite en el cluster ATPS.
- **Programación.** Ha sido necesario programar módulos de jerarquía de memoria de Simics para conseguir los resultado deseados y los scripts necesarios para realizar los experimentos.
- **Caracterización de CloudSuite.** Esta es la tarea que realiza el objetivo principal del proyecto, dentro de la cual se encuentra el diseño y ejecución de los experimentos necesarios para caracterizar la CloudSuite y la recopilación y análisis de los resultados.
- **Documentación.** Esta parte se corresponde con la redacción de la memoria en LaTeX. También a la redacción de un tutorial de iniciación a Simics que permita ayudar a otros en futuros proyectos.

Durante el desarrollo del proyecto se llevaron a cabo todas las tareas planeadas y el trabajo se finalizó en la fecha prevista.

A.2. Esfuerzo invertido

Este proyecto a conllevado la inversión de un total de unas 700 horas. En la gráfica A.2 se presenta el porcentaje de horas dedicadas a cada tarea. Cómo ya indicábamos la parte más importante del proyecto, la caracterización de la CloudSuite, ha requerido la mayor parte del tiempo seguida de la formación. En la tablaA.1 se muestra de manera más detallada la cantidad de horas invertidas en las actividades que componen cada tarea.

A.3. Estimación horas CPU

Para los experimentos realizados se ha consumido un total de aproximadamente 949 horas de CPU, sin embargo si añadimos también todos aquellos experimentos

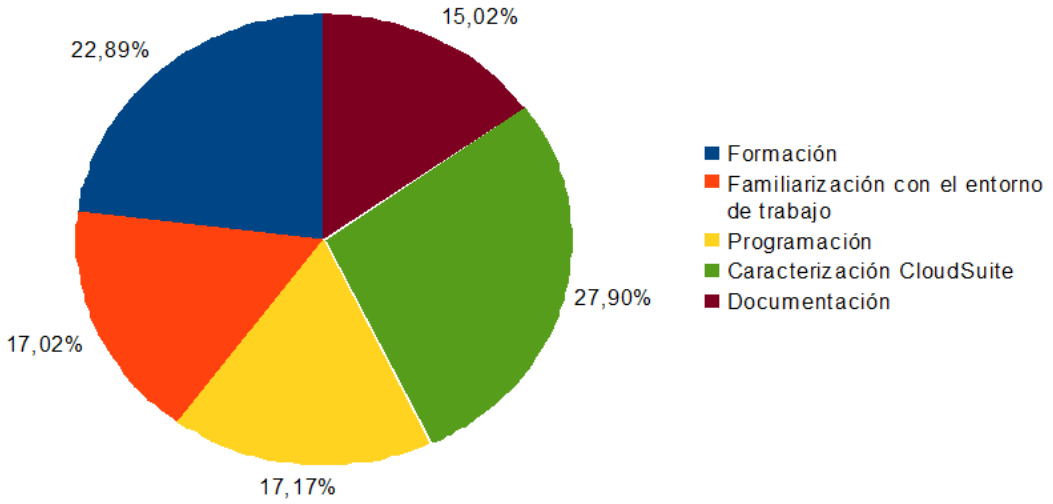

FIGURA A.2: Distribución del tiempo invertido en el proyecto.

que por errores, u otras razones fueron descartados esta estimación, como sucede en todos los trabajos de investigación de arquitectura de computadores, podría hasta triplicarse.

Estas horas de simulación corresponden a 550 segundos de aplicación real. Haciendo los cálculos apropiados vemos que la máquina virtual funciona a una velocidad equivalente de 1'59 MIPS para el algoritmo MC (múltiples caches) y 1'17 MIPS para el algoritmo CCA (cache completamente asociativa). Dado que los procesador físicos del cluster ATPS tiene una frecuencia de 3Ghz, y podemos suponer que sostienen un ritmo de 2 instrucciones/ciclo, lo cual correspondería a 6 GIPS, podemos apreciar un *slowdown* de 3770 y de 5128, respectivamente.

A.4. Problemas encontrados

El primer problema con el que nos encontramos es la falta de una documentación para iniciarse en Simics y para resolver dudas que iban surgiendo en su uso, ya que el manual de uso del simulador resulta insuficiente.

Uno de los problemas generales de los trabajos de simulación en arquitectura es que estas son muy costosas en tiempo, algunas costaban varios días, así que cualquier error en la simulación conlleva retrasos considerables. Una punto débil del algoritmo MC es que en caso de fallo el algoritmo recorre todo el conjunto (de 8 bloques), para mejorarlo podría proponerse el uso de un algoritmo que con poco coste nos dijera que un bloque *no* está en la cache. Una propuesta podría ser el uso de un filtro de Bloom [Blo70], una estructura de datos probabilística concebida

Tarea	Número de horas
Formación	**160**
Estado del arte de Simuladores	71
Estado del arte de las cargas de trabajo	33
Herramientas para el análisis	56
Familiarización con el entorno de trabajo	**119**
Primeros usos de Simics	59
Iniciación y configuración CloudSuite	60
Programación	120
Programación módulos caches	90
Programación Scripts	30
Caracterización CloudSuite	**195**
Diseño y ejecución de experimentos	75
Análisis de Resultados	120
Documentación	**105**
Tutorial Simics	43
Memoria	62
Total	**699**

Tabla A.1: Horas dedicadas a cada tarea del Proyecto.

por Burton Howard Bloom en 1970 que se usa para saber si un elemento forma parte de un conjunto. El test determina con seguridad si un elemento *no* está en el conjunto, o de forma insegura si lo está (o quizás no, es un falso positivo).

Otro de los problemas encontrados para la realización del proyecto es que el cluster utilizado, ATPS, dejó de funcionar dos veces: en diciembre y en agosto. Está última coincide con las tareas de mantenimiento de la universidad pero su restablecimiento se retrasó debido a problemas en Danae, otro cluster del que depende, administrado por el Dpto. de Informática e Ingeniería de Sistemas. Además coincidió con la etapa de mayor intensidad de experimentos y algunos de ellos fueron interrumpidos y tuvieron que repetirse después.

Apéndice B

Productividad en Simics

Uno de los objetivos de este proyecto era producir documentación para posteriores proyectos, ya sean de grado, máster o investigación que requieran Simics como plataforma de simulación. Y de esta manera facilitar y agilizar la fase de formación de estos proyectos. En este tutorial se guiará la ejecución de Simics y sus principales funciones en el cluster ATPS del grupo Gaz de la Universidad de Zaragoza. El tutorial está basado y hace referencias al "Simics User Guide For Unix".

GRUPO DE ARQUITECTURA DE COMPUTADORES DE LA UNIVERSIDAD DE ZARAGOZA

Tutorial de Simics

Este tutorial guiará a través de los primeros pasos para la ejecución de Simics y su configuración en el cluster ATPS del grupo Gaz de la Universidad de Zaragoza.

1. DIRECTORIOS DE TRABAJO

Primero debemos crear un directorio de trabajo. Como estamos trabajando en ATPS para que el directorio de trabajo sea visible por los nodos debemos crearlo en: /export/home/iduser Siendo iduser vuestro nombre de usuario de atps, podemos crear una carpeta llamada common y crear alli el workspace, siendo su ruta: /export/home/iduser/common/workspace Para configurar este workspace debemos ejecutar workspace-setup que se encuentra en el directorio de instalación de Simics (/usr/local/pkg/simics-3.0.31/bin/workspace-setup en atps). Para trabajar más cómodamente podemos linkar esta carpeta commons en nuestro home usando el comando "ln -s".

Además del workspace necesitaremos tres directorios más: Checkpoints, Craffs y tmp, ya que los archivos que escribiremos allí van a ser bastante grandes se crearan en nuestra carpeta de export/scratch/users/iduser. El directorio scratch tiene gran capacidad pero no realiza copias de seguridad, así que tenemos que tener cuidado y encargarnos de realizarlas.

2. CONFIGURACIÓN ATPS

Veámos ahora cómo hay que configurar ATPS para ejecutar Simics. Primero hay que modificar $HOME/.software y añadir la palabra "simics". Además, en el .profile hay que indicar dónde buscar la licencia, escribimos: LM_LICENSE_FILE=1726@atps.cps.unizar.es Para que estos cambios tengan efecto hay que salir y acceder de nuevo a atps.

Si al ejecutar Simics la ventana que muestra la máquina target (máquina que estamos simulando) no se abriera, se puede probar modificando .profile. Para ello hay que comentar

las 5 líneas de código que aparecen tras el comentario #who i am (aparece la palabra DISPLAY en ellas, así que son fáciles de localizar).

3. INICIAR EL SIMULADOR

Hay que tener instalado un sistema operativo en la máquina que emulamos. En este caso usaremos una máquina SPARC que ejecute Solaris.

Los ficheros system-01.disk.craff, system-sol10.disk.craff, abisko-sol10.state y abisko-sol10.run.simics a los que se hace referencia a continuación se proporcionan en el DVD adjunto:"Ficheros Tutorial Simics".

Primero copiamos system-01.disk.craff y system-sol10.disk.craff en el directorio que habíamos creado para los craffs (/export/scratch/users/iduser/craffs) Los archivos .craff son copias de disco duro (no confundir con los checkpoints) y más adelante veremos que se pueden crear para reutilizar datos de una simulación y poder usarlos para crear checkpoints con otras configuraciones. Después copiamos abisko-sol10.state y abisko-sol10.run.simics en nuestro (..)/workspace/targets/serengenti

Tenemos que modificar las tres líneas que contienen paths en abisko-sol10.state:
Checkpoint_path apuntará a este mismo directorio ((..)/workspace/targets/serengenti) Los otros path tienen que apuntar a los craffs que hemos copiado antes, así que hay que escribir la ruta completa.

Los parámetros para configurar la máquina se encuentran en abisko-sol10.run.simics, en este caso se han añadido los siguientes parámetros:
$num_cpus=1 (1 procesador)
$megs_per_cpu=1024 (1GB de memoria por procesador)
$cpu_class = ultrasparc-iii-plus (máquina con un thread por cpu, ultrasparc-iv tiene dos threads)

Finalmente ejecutamos:
$ simics -stall -x abisko-sol10-run.simics
Con este comando le estamos indicando que ejecute el script abisko-sol10-run.simics, "-stall" indica que queremos que las transacciones de memoria se envíen a la jerarquía de memoría que haya conectada y "-x" que el fichero de entrada que le pasamos es un script.

Los comandos básicos de Simics son: c y ctrl+c, continuar la ejecución del target y detenerla, respectivamente. Para salir de Simics se usa q o exit.

Antes de iniciar la simulación del target hay que indicarle a Simics dónde guardar los archivos temporales, si no tendremos problemas más adelante a la hora de copiar archivos en el target, guardar checkpoints y craffs. Para ello antes de darle a continuar (c) escribimos:
prefs->swap-dir=/export/scratch/users/userid/tmp

Depués introducimos c, y entonces el sistema operativo solaris se cargará. Estará listo cuando aparezca el prompt (#).

Como hemos ido apuntando tenemos dos máquinas, la simulada (target) y la real (host). Es posible la comunicación y la transferencia de archivos, y gracias a los archivos de configuración proporcionados para acceder a los archivos almacenados en el host(atps) desde el target(sparc-

solaris) solo hay que ejecutar en el target:
 # mount /host
Se habrá creado una carpeta host que enlazará con nuestra máquina, por ejemplo para llegar a nuestra home la ruta será /host/home/userid.

4. Creación de checkpoints

Los checkpoints nos permiten volver a un mismo punto después sin necesidad de arrancar la máquina de nuevo. Además, también se guarda el contenido del disco. En el punto que deseemos de ejecución, hacemos ctrl+c en el terminal del simulador y ejecutamos:
simics> write-configuration ruta_check_point/micheckpoint.check

Si queremos iniciar Simics desde ese checkpoint ejecutaremos:
$simics -stall -c ruta_check_point/micheckpoint.check
Podemos ver que usamos el flag "-c" cuando iniciamos Simics desde un checkpoint.

Tambien se puede iniciar Simics y desde alli el checkpoint con:
simics> read-configuration ruta_check_point/micheckpoint.check

Hay que tener mucho cuidado al organizar los directorios en nuestra carpeta de checkpoints porque si cambiamos un checkpoint de lugar no funcionará y habrá que modificar los path de sus archivos.
Otro aspecto a tener en cuenta sobre los checkpoints es que son incrementales. Es decir, si inicias el sistema desde un checkpoint (chk1) y en un punto creas otro checkpoint (chk2), para iniciar el sistema con chk2, chk1 será necesario.

5. Comandos útiles

Aquí tenemos algunos comandos que pueden resultarnos útiles en Simics.

- **list-modules**: Lista todos los módulos que pueden ser cargados en Simics, indicando los que ya lo están.
- **run-command-file**: para ejecutar scripts. Más adelante hablaremos de estos scripts, pueden tener código en Python y comandos de Simics.
- **list-objects**: lista todos los objetos así como información de su clase.
- **output-file-start y output-file-stop**: para guardar la salida de Simics (no del target) en un fichero.
- **help** : Comando de ayuda, si se ejecuta help y un objeto proporciona toda la información sobre ese objeto: sus atributos, sus comandos, su clase...
- **print-time -all**: Muestra para cada procesador el número de instrucción, de ciclo y el tiempo en segundos en el que se encuentra. Si queremos ver los ciclos de un procesador en concreto podemos ejecutar cpu0.print-time.

6. CREACIÓN DE CRAFFS

Los Craffs son similares a los checkpoints pero permiten guardar solamente el estado persistente de una máquina, por ejemplo, los datos que permanecen cuando la máquina está apagada (CRAFF = Compressed Random Access File Format). Normalmente esto quiere decir las imagenes del disco, la memoria flash o el contenido NVRAM. De forma muy similar a los checkpoints se guardan y se cargan con los comandos "save-persistent-state path" y "load-persistent-state path" respectivamente.

¿Para que nos puede ser útil? Copiar grandes archivos desde el host hasta el target puede ser costoso en tiempo, para ello una vez copiados puede guardarse un checkpoint para volver al mismo punto. Sin embargo si cambiamos la configuración de la máquina, por ejemplo 2 cpus en vez de 1 cpu, tendríamos que volver a realizar la operación. En este caso Simics nos permite cargar el craff sobre la nueva configuración, como resultado los ficheros anteriores estarán ya cargados en el target.

7. CREACIÓN DE SCRIPTS

Los scripts de Simics pueden llevar tanto comandos Simics como código Python. Las lineas de código Python deben ir precedidas del carácter , y si algún comando de Simics quiere ser invocado en Python hay que usar "@run_command". Veamos algún ejemplo de código Python en Simics:

simics> @print "This is a Python line"
This is a Python line
simics> @if SIM_number_processors() > 1:
..... print "Wow, an MP system!"
..... else:
..... print "Only single pro :-("
.....
Wow, an MP system!
simics> @run_command("print-time")
processor steps cycles time [s]
cpu0 27828281475 27828281475 371.044

El propósito de invocar un comando de Simics en Python es la potencia de este lenguaje, lo cual nos permitiría, por ejemplo, ejecutar un bucle con comandos de Simics. En el siguiente script podemos ver un ejemplo:

```
prefs->swap-dir=/export/scratch/users/iduser/tmp
#Cargamos checkpoint
read-configuration /export/scratch/users/iduser/checkpoints/mi_checkpoint
@sizeKB=8
@numlines=(sizeKB*1024)/64
```

```
#configuracion caches
#=======================================
@cache = pre_conf_object('cache', 'g-cache')
@cache.cpus = conf.cpu0
@cache.config_line_number = numlines
@cache.config_line_size = 64
@cache.config_assoc = 8
@cache.config_virtual_index = 0
@cache.config_virtual_tag = 0
@cache.config_write_back = 0
@cache.config_write_allocate = 1
@cache.config_replacement_policy = 'lru'
@cache.penalty_read = 0
@cache.penalty_write = 0
@cache.penalty_read_next = 0
@cache.penalty_write_next = 0

#Add Configuration
@SIM_add_configuration([cache], None);
#=========================================
#Timing Model
@conf.cpu0_mem.timing_model= conf.cache
#Ejecucion
@time=0
@run_command("cd /home/iduser/experimentos")
@for x in range(0,100):
        file=open("muestra"+str(sizeKB)+"time"+str(time),'a')
        time=time+10
        file.write("Numero inicial de instrucciones: \n")
        file.write(str(conf.cpu0.steps))
        run_command("c 200000000")
        file.write("\nNumero final de instrucciones: \n")
        file.write(str(conf.cpu0.steps))
        file.write("\nEstadisticas instrucciones \n")
        file.write("\nCpu0")
        file.write("\nOperaciones de lectura: "+str(conf.cache.stat_data_read))
        file.write("\nFallos en operaciones de lectura: "+str(conf.cache.stat_inst_data_
        file.close()
        run_command("cache.reset-statistics")
        run_command("cache.reset-cache-lines")
exit
```

Los dos primero comandos son de Simics, indicamos dónde guardar los archivos temporales y cargamos el checkpoint. A continuación ejecutamos dos instrucciones de Python, las cuales

modifican unas variables que nos permitirán configurar la simulación.

En el siguiente conjunto de instrucciones configuramos un objeto llamado “cache” que se declara como objeto de la clase “g-cache”. Podemos ver como las variables pueden ser usadas para la configuración u otros objetos (cpu0 a través de conf.cpu0).

Finalmente llamamos a la función SIM_add_configuration() que añade el objeto cache a la configuración de Simics. Esta función forma parte de la API de Simics que recoge todas las instrucciones que permiten la comunicación entre Simics y Python. Una vez que el objeto forma parte de la configuración de Simics lo conectamos con el espacio de memoria de la cpu0.

En este punto la configuración de la memoria cache a terminado y procedemos a la ejecución, para guardar los resultados de nuestros experimentos en una carpeta determinada podemos usar el comando *cd path* como en linux. El ejecutarlo a través de Python en vez de como simplemente un comando Simics puede permitirnos cambiar la ruta a través de variables, por ejemplo.
Y finalmente tenemos un bucle de 100 iteraciones que crea un fichero, escribe atributos de objetos de Simics en él, ejecuta la simulación por un número determinado de instrucciones. Tras ello escribe los atributos de estadísticas de lecturas de la cache, cierra el fichero y ejecuta los comandos de Simics que inicializan la cache y sus estadísticas. Al acabar el bucle se ejecuta exit para salir del simulador.

Los scripts pueden ser ejecutados tanto desde el propio Simics:
simics> run-command-file script
Cómo al lanzar Simics:
$ simics -stall -x script

8. SOBRE ESTE TUTORIAL

Este tutorial fue parte del proyecto de fin de carrera “Caracterización de instrucciones en aplicaciones de cloud” presentado en Septiembre de 2014 por la alumna de Ingeniería Informática, Alba Pedro Zapater. El objetivo de la creación de este tutorial fue producir documentación para posteriores proyectos, ya fueran de grado, máster investigación que requieran Simics como plataforma de simulación. Y de esta manera facilitar y agilizar la fase de formación de estos proyectos.

Apéndice C

Módulo G-Cache

En este apéndice vamos a ampliar la información sobre el módulo g-cache que ya presentábamos en el apartado 4.3 y sobre los algoritmos que hemos programado.

C.1. Más sobre G-cache

Para el estudio tanto de g-cache como de la simulación de caches en general se acudió al capitulo 18 del "Simics User Guide For Unix" y al código de g-cache que se encuentra dentro del directorio de instalación de Simics en [simics]/src/extensions. G-cache nos permite simular desde una cache sencilla, definiendo su tamaño de bloque, número de bloques, asociatividad, política de reemplazo, si es copy-back, si es write-allocate y ciclos de penalización por escrituras y lecturas, hasta jerarquías más complicadas. Por ejemplo la de la figura C.1, en la que dos niveles de cache son simulados, y el primer nivel está dividido en cache de instrucciones y cache de datos. G-cache también permite tanto conectar una memoria cache a varias CPUs cómo diseñar un sistema multiprocesador con un protocolo de coherencia MESI. G-cache nos proporciona además las estadísticas sobre el número total de transacciones, número de lecturas, de escrituras, de instrucciones, número de fallos en cada una de estas categorías, número de operaciones en copy-back, y número de invalidaciones, y cambios de estado para el protocolo MESI. Podemos ilustrar g-cache a través de sus diagramas de estado, para ello vamos a usar como ejemplo el sistema de la figura C.2. En ella podemos observar dos niveles de cache, con protocolo de coherencia MESI entre las caches de nivel 2. De este sistema podemos obtener cuatro diagramas de estados: Transiciones por eventos internos de las caches de nivel 2 (C.3), transiciones por eventos externos, correspondientes al protocolo MESI entre caches del nivel 2 (C.4), transiciones por eventos internos de las caches de nivel 1 (C.5) y por último, transiciones por eventos externos,

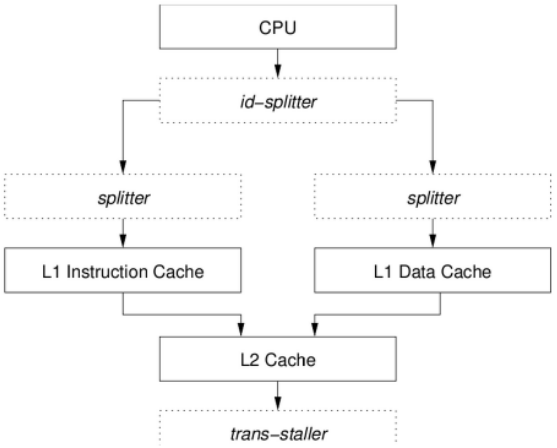

FIGURA C.1: Jerarquía de caches.

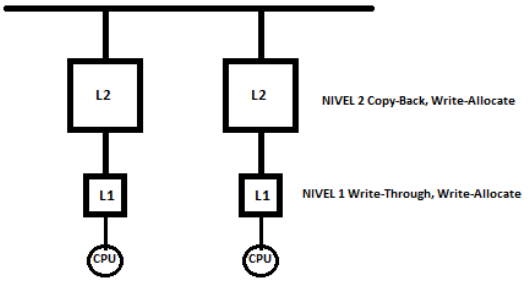

FIGURA C.2: Ejemplo Sistema de Caches multiprocesador con MESI.

correspondientes al protocolo MESI desde caches del nivel 2 a sus caches del nivel 1 (C.6) Veamos los eventos que aparecen en estos diagramas:

- **Internos**:Son aquellos que derivan de los fallos (miss) y aciertos (hits) en escritura (write) y lectura (read) de bloques en la cache o del reemplazo de un bloque. En los diagramas aparecen como rh,wh,wm,rm y rpl.
- **Externos**: Son aquellos que se reciben desde otras caches.
  - inv: Invalidación del bloque.
  - rB: Otra cache va a leer ese bloque.

Y sus acciones asociadas:

- **Mp(write/read, x)**: Envía una lectura o escritura de ese bloque a memoria principal (Mp), a su cache de nivel 2 (Mc2) o a su cache de nivel 1 (Mc1)

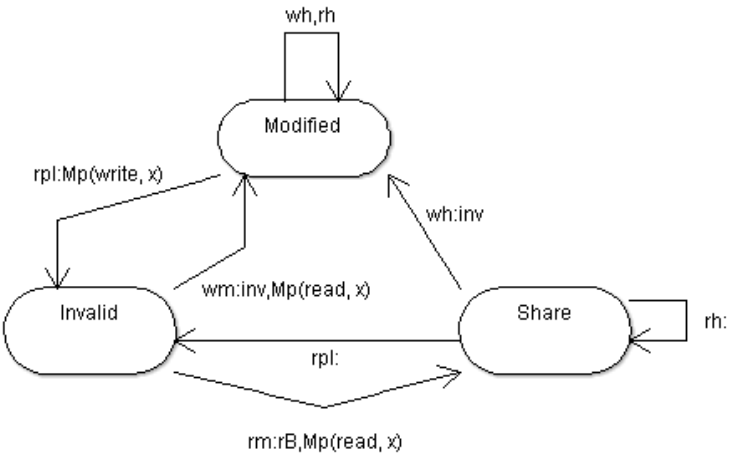

FIGURA C.3: Diagrama de Estados Cache Copy-Back Nivel 2 debido a eventos internos.

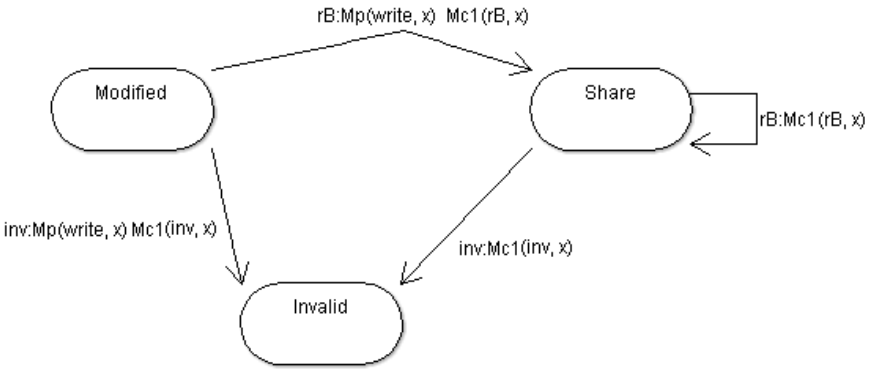

FIGURA C.4: Diagrama de Estados Cache Copy-Back Nivel 2 debido a eventos externos.

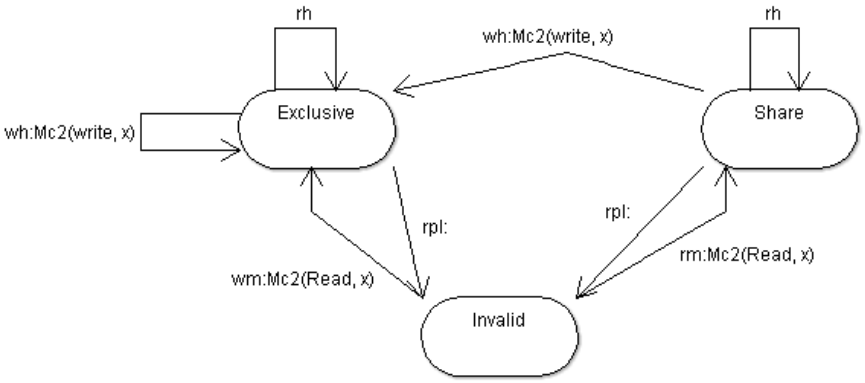

FIGURA C.5: Diagrama de Estados Cache Write-Through Nivel 1 debido a eventos internos.

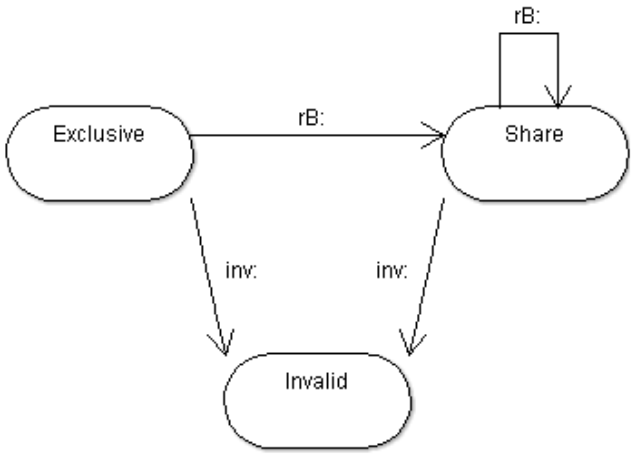

FIGURA C.6: Diagrama de Estados Cache Write-Through Nivel 1 debido a eventos externos.

- **inv, rB**: Envía estos eventos a través del Bus para que lo reciban el resto de caches de nivel 2.

C.2. Algoritmos

Basándonos en el código original del módulo g-cache hemos programado nuestras propias versiones para obtener con eficiencia el comportamiento de múltiples configuraciones de una cache de instrucciones.

C.2.1. Algoritmo para múltiples caches (algoritmo MC)

Como ya hemos descrito en el apartado 4.2.1, la idea de este algoritmo es obtener en una sola simulación la tasa de fallos para distintos tamaños y asociatividades. Para ello hemos programado un algoritmo de reemplazo y aplicado algunas modificaciones al módulo g-cache.

Las principales estructuras de datos del algoritmo son:

- **Vector de punteros a bloque (`Vmru`)**: Un vector con dimensión el número de conjuntos, cada componente apunta al bloque MRU de cada conjunto.
- **Lista LRU dinámica**: Cada bloque apunta al siguiente bloque de su conjunto en orden LRU, el último (el LRU) apunta nulo. (`bloque->sig`, el puntero al siguiente bloque en el seudocódigo)
- **Vector de aciertos (`H`)**: Un vector de contadores con dimensión la asociatividad. Cada componente recoge el número de aciertos que se han dado en

esa posición en la lista LRU. Por ejemplo, si el acierto se ha producido en la posición 2 de la lista LRU, se incrementará en uno el elemento 2 del vector. Como resultado al final de la ejecución tenemos el vector con el número de aciertos en cada posición LRU.

Aqui podemos ver el algoritmo en seudocódigo:

```
//Búsqueda de bloque
set= obtener_set (mem_op) //Obtenemos el número de conjunto al que pertenece el bloque de
                          // la operación de memoria
bloque=Vmru[set]
numbloque=-1
posacierto=0
ant=null
Si bloque != invalido y bloque==obtener_bloque(mem_op) entonces
{
        numbloque=bloque.num

}
Si (bloque != invalido y bloque->sig!=NULL) {

        hacer{
                    bloque=bloque->sig
                        posacierto=posacierto+1;
                        ant=bloque;
                        bloque=ant->sig;
                        Si bloque==obtener_bloque(mem_op){
                        numbloque=bloque.num
                        }
        }mientras((numbloque==-1) y (bloque->sig!=NULL) && (bloque != invalido));
        }
    Si (numbloque!=-1){//Acierto
        H[posacierto]=H[posacierto] + 1
        }
        if (ant==null){//Primer elemento invalido, o solo un elemento o el primer elemento
era el acierto
        devolver numbloque}
        sino {
        //Hay que reordenar la lista LRU, en caso de fallo el primer elemento es el bloque víctima,
en caso de acierto el primer bloque es el referenciado

                ant->sig=bloque->sig;
                bloque->sig=Vmru[set]; //El bloque MRU anterior
                Vmru[set]=bloque
                devolver numbloque
```

En caso de fallo, cuando el algoritmo principal de g-cache solicite el bloque víctima será aquel que este en primer lugar en la lista, ya que anteriormente lo hemos modificado, y será reemplazado por el nuevo bloque. Las estadísticas se obtienen posteriormente procesando los datos obtenidos. El número total de instrucciones y los fallos de la cache simulada ya los proporciona g-cache. En la tabla C.1 tenemos un ejemplo de las estadísticas que se pueden obtener a partir del vector de aciertos `H` para una cache de asociatividad 8 y tamaño “Size”.

Tamaño	Asociatividad	Aciertos
Size/8	1	H[0]
Size/4	2	H[0] + H[1]
Size/2	4	H[0] + H[1] + H[2] + H[3]
Size	8	Suma(H)

Tabla C.1: Relación de tamaño, asociatividad y aciertos para una ejecución con el algoritmo MC.

C.2.2. Algoritmo para cache completamente asociativa (algoritmo CCA)

Para simular una cache completamente asociativa (un sólo conjunto) de forma clásica, para cada referencia se visitan los bloques en orden, desde el mas al menos reciente. Tras encontrar el bloque buscado se actualiza la lista y se incrementa el contador de aciertos. En caso contrario, tras haber visitado todos los bloques, se reemplaza el bloque víctima (el bloque LRU), se ajusta la ordenación y se incrementa el contador de fallos.

Este algoritmo precisa recorrer y ordenar una lista cuyo tamaño medio coincide con la distancia media de acierto, que suele ser del orden de las decenas. Esto supone un alto coste en tiempo, por lo cual vamos proponemos una alternativa mas rápida. Nuestra propuesta se basa en utilizar una estructura de datos auxiliar, en forma de una cache de correspondencia directa (puede verse como una estructura *hash* de aceleración, `caux`, en el seudocódigo). Los bloques deben estar ordenados por orden de uso, pero esta ordenación puede conseguirse de forma explícita, como en el algoritmo anterior, o de forma implícita, usando marcas de tiempo, como haremos ahora. Cada bloque tiene una marca de tiempo que indica en que ciclo fue usado por ultima vez.

Las características principales del algoritmo CCA son:

- El tamaño de la cache auxiliar debe de ser al menos del tamaño de la cache principal, pero es recomendable que sea lo más grande posible.
- Para evitar el coste de gestión de listas ordenadas, la ordenación LRU se ha implementado con marcas de tiempo en cada bloque, que consignan el ciclo en el que ha sido referenciado por última vez.
- Un acierto en la cache auxiliar garantiza acierto en la cache principal, aunque un fallo no determina si el bloque está o no en la principal. Sin embargo, debido a la localidad temporal y espacial, la mayoría de los aciertos se producen en los bloques de la cache de correspondencia directa, como podemos observar en la gráfica de las 3Cs, ver figura 4.2.

- En caso de fallo en la cache auxiliar, se recorre la cache principal. Si se encuentra el bloque, se produce un acierto, y la cache auxiliar debe actualizarse para contenerlo.
- El mayor coste es para los fallos con cache llena, ya que hay que recorrer toda la cache. En este mismo recorrido se descubre el bloque víctima, es decir aquel cuya marca de tiempo sea más antigua.

Aquí podemos ver el algoritmo en seudocódigo:

```
//Búsqueda de bloque
                bloquevictima=-1
                lru_tiempo=max_enteros
                set=obtener_set_caux(mem_op)
                numbloque=caux[set]
                Si (numbloque!=-1) {
                        Si(cache[numbloque]==obtener_bloque(mem_op)){
                                devolvemos numbloque
                        }
                }
                Desde i=0 a numerobloques{
                        Si (cache[i]==invalido){//fallo, hemos llegado a los
    inválidos
                                bloquevictima=i;
                                devolvemos -1;
                        }

                        Si(cache[numbloque]==obtener_bloque(mem_op)){//
    acierto
                                devolvemos numbloque
                        }
                Si (cache[i].marcadetiempo < lru_tiempo) {//Busca LRU
                                lru_bloque = i;
                                lru_tiempo = cache[i].marcadetiempo;
                        }

                }
                //Fallo, habrá un bloque victima válido: lru_bloque

                set=obtener_set_caux(lru_bloque)
                numbloque=caux[set];
                Si (numbloque==lru_bloque) {
                     caux[set]=-1; //Invalidamos
                        }
                        bloquevictima=lru_bloque;
                        devolver -1; //Es fallo, la cache está llena, devolvemos el
    lru.
```

```
//Actualización caux tras fallo o acierto
//bloque es el bloque que acabamos de referenciar
        bloque.marcadetiempo= obtener_ciclo_CPU();
                set=obtener_set_caux(bloque);
                caux[set]=bloque.numero;//Actualizamos la cache auxiliar
```

Se ha observado una mejora en el tiempo de ejecución de un 90,52 % al usar este algoritmo frente a usar el algoritmo original de LRU de g-cache que esta basado en una lista única con marcas de tiempo.

Apéndice D

Simulaciones CloudSuite

En este Apéndice vamos a presentar dónde y cómo se han llevado a acabo las simulaciones de la CloudSuite.

D.1. Cluster ATPS

Los experimentos han sido realizados en el cluster ATPS. El cluster ATPS es una infraestructura de computación de altas prestaciones financiada por el grupo de Arquitectura de Computadores de la Universidad de Zaragoza (gaZ). ATPS se usa principalmente para simular modelos funcionales y temporales a nivel micro-arquitectura (procesadores, caches y redes de interconexión). ATPS es un cluster que se usa fundamentalmente en modo de productividad. Se lanzan múltiples experimentos (variaciones de un modelo con distintos parámetros) que se ejecutan independientemente en las máquinas del cluster. El cluster consta de 6 chasis de dos tipos:

- 3 chasis 1U, cada uno con 2 nodos 2x Intel Xeon X5365 (4Cores, 3.00 GHz, 8 MB L2), en total 6 nodos, uno se dedica al front-end y los otros dedicados a computación. En cada nodo hay pues un total de 8 cores compartiendo 16 GB RAM.
- 3 chasis 2U modelo Superserver SYS-6026TT-TRF, cada uno con 4 nodos 2xIntel Xeon X5650 (Westmere, 6 Cores, 2.67 GHz, 12 MB L3), en total 12 nodos, todos dedicados a computación. En total en cada nodo hay 12 cores (24 threads) compartiendo 48 GB RAM.

En la figura D.1 podemos ver el esquema de conexiones de red que permite simplificar la administración de los distintos equipos, ya que todos los nodos están conectados al front-end a través de la misma dirección IP.

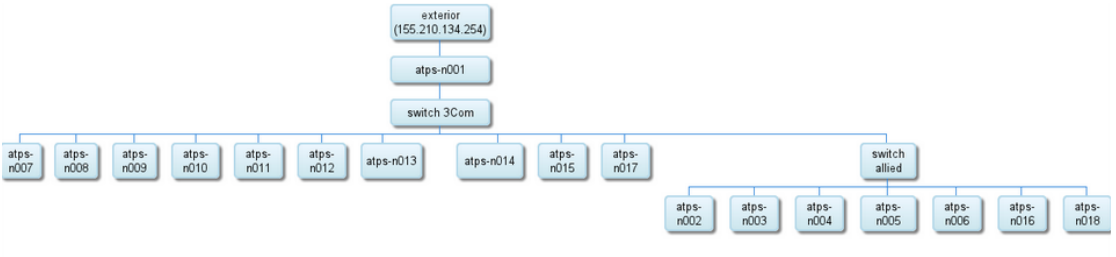

FIGURA D.1: Red de conexiones atps.

D.2. Condor

Condor es un sistema de gestión de carga para tareas de computación intensivas que soporta un gestor de colas, con políticas de planificación de ejecución, esquema de prioridades, monitorización y gestión de recursos. Es decir, desde el front-end se lanzan los trabajos a través de Condor que se encarga de distribuirlos en los nodos. A continuación podemos ver un script muy sencillo para lanzar un trabajo, aunque Condor permite muchas más opciones de configuración para necesidades más sofisticadas.

```
# Example submit file for vanilla job

Universe        = vanilla
Executable      = hello_world.sh

input    = /dev/null
output   = hello.out
error    = hello.error

Queue
```

Lo habitual es lanzar las tareas a los nodos de ATPS a través de condor, sin embargo al compartir la máquina con un entorno de producción que limitaba los recursos que podíamos utlizar, pasamos a programar shell scripts para lanzar manualmente los trabajos en el número limitado de nodos que nos fueron asignados.

D.3. Scripts

En esta sección detallamos los scripts usados para las simulaciones a las que se hace referencia en el apartado 4.4.

D.3.1. Shell Scripts

Un script por cada aplicación. En el ejemplo ilustramos el lanzamiento de una simulación de la aplicación Cassandra.

```
size=8
for ((i=0;i<6;i++))
do
 oldsize=$size
 size=‘expr $size \* 2‘
 string="s/@sizeKB=$oldsize/@sizeKB=$size/"
 sed -i $string /home/albapz/common/workspacenodos/scripts/OptCassan4cpu
 echo $size
 ./simics -stall -no-win -x /home/albapz/common/workspacenodos/scripts/
    OptCassan4cpu
done
```

El script anterior permite lanzar 6 ejecuciones de Simics a través de un bucle. En cada iteración con el comando “sed” se modifica el script de Simics “OptCassan4cpu”. Concretamente modifica la variable que define el tamaño de la cache, doblando este valor en cada iteración. Tras esto Simics ejecuta el script mencionado. Al lanzar Simics se le indica a través del flag `-no-win` que desactive la apertura de ventanas externas (las del target o cualquier otra externa).

D.3.2. Simics Scripts

Los scripts de Simics pueden contener comandos de Simics y/o código Python (se indica con el símbolo `@` al principio de la linea de código). Al lanzar Simics con el flag `-x` indicamos que tiene que ejecutar el script que se pasa como parámetro. Veamos un ejemplo:

```
prefs->swap-dir=/export/scratch/users/albapz/tmp
#Cargamos checkpoint
read-configuration /export/extra/data/trazas/parsa/cloudsuite/images/
    cassandra/4cpu/4s-4gb-2c-4gb
@sizeKB=8
@numlines=(sizeKB*1024)/64
#configuracion caches
istc-disable
@conf.server_cpu0.instruction_fetch_mode ="instruction-fetch-trace"
@conf.server_cpu1.instruction_fetch_mode ="instruction-fetch-trace"
@conf.server_cpu2.instruction_fetch_mode ="instruction-fetch-trace"
@conf.server_cpu3.instruction_fetch_mode ="instruction-fetch-trace"
@conf.client_cpu0.instruction_fetch_mode ="instruction-fetch-trace"
@conf.client_cpu1.instruction_fetch_mode ="instruction-fetch-trace"

#=============================================
## Transaction staller for memory
#=============================================
@staller0 = pre_conf_object('staller0', 'trans-staller')
```

```
@staller0.stall_time = 0
#=================================================================
#=========================================
## L1 - Instruction Cache : L1 Inst0
@ic0 = pre_conf_object('ic0', 'g-cache')
@ic0.cpus = conf.server_cpu0
@ic0.config_line_number = numlines
@ic0.config_line_size = 64
@ic0.config_assoc = 8
@ic0.config_virtual_index = 0
@ic0.config_virtual_tag = 0
@ic0.config_write_back = 0
@ic0.config_write_allocate = 1
@ic0.config_replacement_policy = 'lruopt'
@ic0.penalty_read = 0
@ic0.penalty_write = 0
@ic0.penalty_read_next = 0
@ic0.penalty_write_next = 0
@ic0.timing_model = staller0

#=========================================
## ID splitter for L1 cache
@id0 = pre_conf_object('id0', 'id-splitter')
@id0.ibranch = ic0
@id0.dbranch = staller0


#=========================================
## L1 - Instruction Cache : L1 Inst1
@ic1 = pre_conf_object('ic1', 'g-cache')
@ic1.cpus = conf.server_cpu1
@ic1.config_line_number = numlines
@ic1.config_line_size = 64
@ic1.config_assoc = 8
@ic1.config_virtual_index = 0
@ic1.config_virtual_tag = 0
@ic1.config_write_back = 0
@ic1.config_write_allocate = 1
@ic1.config_replacement_policy = 'lruopt'
@ic1.penalty_read = 0
@ic1.penalty_write = 0
@ic1.penalty_read_next = 0
@ic1.penalty_write_next = 0
@ic1.timing_model = staller0

#=========================================
## ID splitter for L1 cache
@id1 = pre_conf_object('id1', 'id-splitter')
@id1.ibranch = ic1
@id1.dbranch = staller0
#=====================================
#Add Configuration
@SIM_add_configuration([staller0,ic0, id0, ic1, id1], None);
#=========================================
#Timing Model
```

```
@conf.server_cpu0_mem.timing_model= conf.id0
@conf.server_cpu1_mem.timing_model= conf.id1
#Ejecucion
@time=0
@run_command("cd /home/albapz/common/workspacenodos/experimients/lruopt/
    cassan")
@for x in range(0,100):
        file=open("cassanopt2cpus"+str(sizeKB)+"time"+str(time),'a')
        time=time+10
        file.write("Numero inicial de instrucciones: \n")
        file.write(str(conf.server_cpu0.steps))
        run_command("c 200000000")
        file.write("\nNumero final de instrucciones: \n")
        file.write(str(conf.server_cpu0.steps))
        file.write("\nEstadisticas instrucciones \n")
        file.write("\nCpu0")
        file.write("\nInstruction Fetch  transactions: "+str(conf.ic0.
    stat_inst_fetch))
        file.write("\nInstruction Fetch  misses: "+str(conf.ic0.
    stat_inst_fetch_miss))
        file.write("\nVector Hits: "+str(conf.ic0.lruopt_hits))
        file.write("\nCpu1")
        file.write("\nInstruction Fetch  transactions: "+str(conf.ic1.
    stat_inst_fetch))
        file.write("\nInstruction Fetch  misses: "+str(conf.ic1.
    stat_inst_fetch_miss))
        file.write("\nVector Hits: "+str(conf.ic1.lruopt_hits))
        file.close()
        run_command("ic0.reset-statistics")
        run_command("ic0.reset-cache-lines")
        run_command("ic1.reset-statistics")
        run_command("ic1.reset-cache-lines")
        conf.ic0.lruopt_hits=[0,0,0,0,0,0,0,0]
        conf.ic1.lruopt_hits=[0,0,0,0,0,0,0,0]


exit
```

En este script podemos destacar:

- El parámetro `-sizeKB` indica el tamaño de la cache a simular. Este parámetro es el que se modifica desde el Shell Script. `numlines` indica el número de bloques.
- Simics usa internamente unas caches software (de datos y de instrucciones) para acelerar las simulaciones, a las que llama STC (Simulator Translation Cache) que evitan que todas las transacciones tengan que pasar por la jerarquía de memoria. Sin embargo en el caso de las instrucciones esto implica estadísticas incorrectas, por lo cual procedimos a su desactivación con el comando `istc-disable`.

- Por defecto, y también para acelerar las simulaciones, Simics no envía las búsquedas de instrucciones a la jerarquía de memoria. Para evitar este comportamiento hay que cambiar el modo de simulación de las cpus a `instruction-fetch-trace`
- `Staller` representa el acceso a la memoria principal, sin embargo en este caso la penalización por acceso a memoria principal es 0.
- Se configuran dos caches de instrucciones, una por cada procesador. En este caso la política de reemplazo `lruopt` corresponde al algoritmo MC descrito en el apartado C.2.1.
- Se declara un `ID Splitter` para cada procesador; este objeto de Simics separa las transacciones de datos de las de instrucciones.
- Todos los objetos declarados se añaden a la configuración de Simics y se conecta cada `ID Splitter` al `timing_model` de los procesadores cuyas caches de instrucciones queremos simular (los procesadores que ejecutan los servidores de interés en cada aplicación). Al `timing_model` puede conectarse un objeto para que tenga acceso a la transacción de memoria, antes de que ésta se ejecute. En nuestro caso ese objeto es nuestra jerarquía de memoria.
- Finalmente comienza la ejecución de la simulación, para ello un bucle en Python ejecuta 100 veces 200 millones de instrucciones (por cada procesador, ya que trabajan en paralelo) y tras cada ejecución se escriben las estadísticas en un fichero, inicializando de nuevo tanto las estadísticas como las caches.

La figura D.2, representa la jerarquía de memoria configurada en el script de ejemplo.

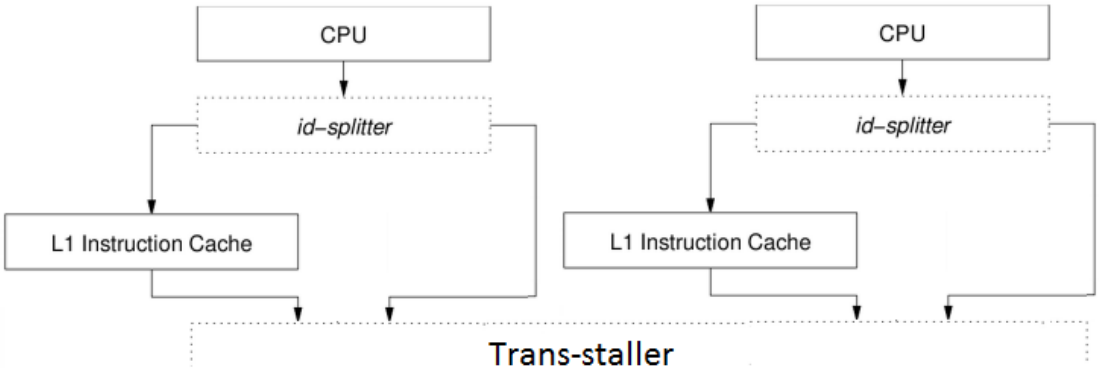

FIGURA D.2: Jerarquía de cache configurada con el script.

Bibliografía

[BKSL08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008. http://parsec.cs.princeton.edu/ [Online; accessed 23-Agosto-2014].

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[Bon07] Jan Lodewijk Bonebakker. Finding representative workloads for computer system design. Technical report, Sun Microsystems, Inc. Mountain View, CA, USA, 2007.

[CGS99] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., 1999.

[CPU06] SPEC CPU2006. https://www.spec.org/cpu2006/, 2006. [Online; accessed 23-Agosto-2014].

[EPF] PARSA EPFL. Cloudsuite official webpage. `http://parsa.epfl.ch/cloudsuite/cloudsuite.html`.

[FAK+12] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Djordje Jevdjic Mohammad Alisafaee, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds. a study of emerging scale-out workloads on modern hardware. In *17th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*, March 2012.

[HP06] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[HS89] M.D. Hill and A.J. Smith. Evaluating associativity in cpu caches. *Computers, IEEE Transactions on*, 38(12):1612–1630, Dec 1989.

[KMHW03] Martin Karlsson, Kevin Moore, Erik Hagersten, and David Wood. Memory System Behavior of Java-Based Middleware. pages 217–228, Anaheim, California, USA, February 2003.

[LVIB11] José María Llabería, Víctor Viñals, Pablo Ibáñez, and Ana Bosquel. *Filtering directory lookups in CMPS.* PhD thesis, Zaragoza, Universidad de Zaragoza, Zaragoza, Ago 2011. http://zaguan.unizar.es/record/6812?ln=es.

[MCE+02] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, Feb 2002.

[MGST70] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, June 1970.

[MSB+05] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, , and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33:92–99, Nov 2005.

[TC] TPC-C. http://www.tpc.org/tpcc/. [Online; accessed 23-Agosto-2014].

[web09] SPEC web2009. http://www.spec.org/web2009/, 2009. [Online; accessed 23-Agosto-2014].