Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Proyecto Fin de Carrera
Ingeniería Informática

Caracterización del comportamiento de la suite PARSEC en la jerarquía de memoria del procesador

Marta Ortín Obón

Directores: María Villarroya Gaudó y Darío Suárez Gracia

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

Curso 2010/2011
Septiembre 2011

A mis padres y mi abuela.

Agradecimientos

Quiero agradecer a mis directores, María y Darío, las oportunidades que me han dado y su ayuda en todo lo que he necesitado, tanto a lo largo de la beca como del proyecto. Gracias a vosotros me he iniciado en la investigación y tengo claro lo que quiero hacer después de terminar la carrera. También les doy las gracias a Víctor y Pablo, que han estado muy involucrados en mi trabajo y me han dado muchas ideas y consejos. Además, agradezco a Chus y Kike todo lo que me han enseñado y su gran ayuda en algunas secciones de este proyecto.

En especial, le quiero dar las gracias a Jorge, que me ha ayudado muchísimo durante los últimos meses. Me he aprovechado de que estabas el despacho de al lado y te he preguntado montones de dudas, pero siempre me has recibido con una sonrisa y me has dedicado todo el tiempo que necesitaba.

También quiero darles las gracias a mis amigos de clase, que han hecho mucho más alegres todos mis días en la universidad. Principalmente, a Sergio, Xandra, Rubén, Sara y Edu, que me han acompañado durante los meses del proyecto y con los he compartido los buenos momentos, las frustraciones, las comidas, las sobremesas, los descansos y, en general, todos los ratos poco productivos que he pasado en el CPS.

Por último, quiero dar las gracias a mis padres por todo su apoyo y a mi abuela, que desde pequeña me ha enseñado a superarme a mí misma.

Caracterización del comportamiento de la suite PARSEC en la jerarquía de memoria del procesador

Resumen ejecutivo

Los simuladores son herramientas fundamentales para el diseño de nuevas arquitecturas de computadores. En este campo, interesa disponer de simuladores detallados que ofrezcan resultados precisos y, al mismo tiempo, utilizar cargas de trabajo realistas que proporcionen conclusiones objetivas. El principal obstáculo para las simulaciones es su alto coste en tiempo y memoria, lo que nos lleva a sacrificar la precisión del simulador o a utilizar aplicaciones demasiado ligeras que resultan poco representativas.

En este proyecto, se ha realizado un estudio del propio simulador y las cargas de trabajo con el objetivo de conseguir simulaciones representativas de una ejecución realista en un tiempo razonable. Nos hemos centrado en la plataforma Virtutech Simics, un simulador de sistema completo ampliamente utilizado, y GEMS, que proporciona módulos para la simulación temporal. Como carga de trabajo hemos seleccionado PARSEC, que ofrece un conjunto representativo de las nuevas aplicaciones paralelas emergentes de memoria compartida.

Se ha analizado el tiempo de simulación con Simics y GEMS buscando cuellos de botella que pudieran ser optimizados. Hemos observado que gran parte del tiempo de simulación recae sobre el módulo de GEMS que se ocupa de la jerarquía de memoria, aunque el tiempo está muy disperso dentro del módulo, dificultando la optimización.

Ante estos resultados, pasamos a estudiar la suite PARSEC centrándonos en su comportamiento en la jerarquía de memorias cache. Cuando se usan estas aplicaciones en investigación, se suelen utilizar entradas de tamaño reducido como aproximación de una entrada nativa porque el tiempo de simulación resulta más conveniente. No obstante, no está demostrado que estas entradas destinadas a simulación sean adecuadas para obtener resultados representativos. Además, existe la creencia popular de que cuanto más complejo es el problema a resolver, mayor presión se ejerce sobre la jerarquía de memoria.

Hemos utilizado herramientas de análisis (profiling) y simulación para obtener distintas métricas de las aplicaciones de PARSEC con sus entradas de diferentes tamaños. Analizando estos resultados, descubrimos que no necesariamente las entradas más grandes presentan mayores tasas de fallos y que la entrada nativa no genera un número de fallos notablemente más elevado que el resto. Estos resultados se han obtenido analizando el comportamiento de las aplicaciones ejecutándolas con un thread sobre un nivel de memoria cache, aunque se presume que las conclusiones seguirán siendo válidas para múltiples threads. La verificación de esta hipótesis queda planteada como trabajo futuro.

Como resultado final del proyecto, hemos realizado una selección de las entradas más representativas de una ejecución nativa que permiten obtener resultados fiables en un tiempo razonable. Para ello se han utilizado diferentes técnicas: ejecución de una sección de la entrada nativa, uso de una entrada de menor tamaño o uso de una nueva entrada distinta de todas las que ya existen. La utilización de estas entradas para las aplicaciones de PARSEC resulta más adecuada que el uso sistemático de una de menor tamaño, ya que permite conseguir resultados más representativos manteniendo un tiempo de simulación razonable.

Contenidos

Índice de figuras v

Índice de tablas ix

1 Introducción 1

1.1 Contexto del proyecto . . . 2

1.2 Objetivos . . . 2

1.3 Organización de la memoria . . . 3

2 Estado del arte 5

2.1 Plataformas y estrategias de simulación . . . 5

2.2 Cargas de trabajo . . . 6

3 La suite PARSEC 9

4 Metodología 11

4.1 Introducción a las métricas utilizadas . . . 11

4.2 Footprint de la memoria . . . 12

4.3 Obtención de los fallos de TLB . . . 13

4.4 Instrumentación del programa utilizando VALGRIND . . . 14

4.5 Estudio de la jerarquía de memoria mediante simulación . . . 15

5 Resumen de resultados 17

5.1 Análisis del tiempo de simulación . . . 17

5.2 Impacto del tamaño de las entradas en la jerarquía de memoria . . . 18

5.2.1 Instruction mix . . . 18

5.2.2 Footprint . . . 18

5.2.3 Fallos de TLB . . . 19

5.2.4 Tasa de fallos en cache y traza temporal . . . 20

5.3 Selección de entradas . . . 24

6 Conclusiones y trabajo futuro 27

6.1 Conclusiones a nivel técnico y trabajo futuro . . . 27

6.2 Conclusiones a nivel personal . . . 28

Bibliografía 29

A Gestión del proyecto **33**
A.1 Gestión del tiempo 33
A.2 Esfuerzo invertido 34
A.3 Problemas encontrados 35

B Análisis del tiempo de simulación: Simics y GEMS **37**
B.1 Tiempo de ejecución de las simulaciones 37
B.2 Distribución del tiempo en los diferentes módulos durante la ejecución de la simulación 38
B.2.1 Tipos de simulaciones realizadas 39
B.2.2 Resultados de la distribución de tiempos 40
B.3 Conclusiones 44

C Detalles de las simulaciones con Simics y GEMS **45**
C.1 Fases de desarrollo de los experimentos. 45
C.2 Diseño de las simulaciones 46
C.3 Ejecución de las simulaciones 50
C.4 Recopilación de resultados 51

D Resultados de la caracterización de PARSEC **55**
D.1 Impacto del tamaño de las entradas en la jerarquía de memoria 55
D.1.1 Instruction mix 55
D.1.2 Footprint 56
D.1.3 Fallos de TLB 57
D.1.4 Tasas de fallos en cache y trazas temporales 57
D.2 Selección de entradas 103
D.2.1 Blackscholes 103
D.2.2 Bodytrack 103
D.2.3 Canneal 104
D.2.4 Dedup 104
D.2.5 Facesim 104
D.2.6 Ferret 104
D.2.7 Fluidanimate 105
D.2.8 Freqmine 105
D.2.9 Raytrace 105
D.2.10 Streamcluster 106
D.2.11 Swaptions 107
D.2.12 Vips 107
D.2.13 X264 107

Glosario **110**

Índice de figuras

4.1 50 %, 90 % y 100 % del footprint de la aplicación blackscholes 13

5.1 Instruction mix de las aplicaciones de PARSEC 18
5.2 Footprint de las aplicaciones de PARSEC 19
5.3 Fallos en el TLB de datos de las aplicaciones de PARSEC. 19
5.4 Fallos por cada mil instrucciones en la cache de datos para blackscholes ejecutado en Intel 21
5.5 Fallos por cada mil instrucciones en la cache de datos para blackscholes ejecutado en Sparc 21
5.6 Traza temporal de fallos en cache para blackscholes con entradas pequeña, mediana y grande 22
5.7 Traza temporal de fallos en cache para blackscholes con entrada nativa 23

A.1 Diagrama de Gantt del proyecto. 33
A.2 Distribución del tiempo en las diferentes tareas del proyecto. 34

B.1 Tiempo de simulación de canneal, fluidanimate y streamcluster con Simics y Simics+GEMS, con 1 y 2 procesadores 38
B.2 Slowdown de las simulaciones de canneal, fluidanimate y streamcluster con Simics y Simics+GEMS, con 1 y 2 procesadores 39
B.3 Distribución del tiempo en los diferentes módulos simulando blackscholes. . . 41
B.4 Distribución del tiempo en los diferentes módulos simulando bodytrack. 41
B.5 Distribución del tiempo en los diferentes módulos simulando canneal. 42

C.1 Fases de realización de experimentos 45

D.1 Número de instrucciones de las aplicaciones de PARSEC con cada una de sus entradas 56
D.2 Número medio de accesos a cada página de memoria de las aplicaciones de PARSEC 57
D.3 50 %, 90 % y 100 % del footprint de las aplicaciones de PARSEC (parte 1) 58
D.4 50 %, 90 % y 100 % del footprint de las aplicaciones de PARSEC (parte 2) 59
D.5 Fallos por cada mil instrucciones en la cache de datos para bodytrack ejecutado en Intel 61
D.6 Fallos por cada mil instrucciones en la cache de datos para bodytrack ejecutado en Sparc 61
D.7 Traza temporal de fallos en cache para bodytrack con entradas pequeña, mediana y grande 62
D.8 Traza temporal de fallos en cache para bodytrack con entrada nativa 63

D.9 Fallos por cada mil instrucciones en la cache de datos para `canneal` ejecutado en Intel . . . 64
D.10 Fallos por cada mil instrucciones en la cache de datos para `canneal` ejecutado en Sparc . . . 65
D.11 Traza temporal de fallos en cache para `canneal` con entradas pequeña, mediana y grande . . . 66
D.12 Traza temporal de fallos en cache para `canneal` con entrada nativa . . . 67
D.13 Fallos por cada mil instrucciones en la cache de datos para `dedup` ejecutado en Intel 68
D.14 Fallos por cada mil instrucciones en la cache de datos para `dedup` ejecutado en Sparc 69
D.15 Traza temporal de fallos en cache para `dedup` con entradas pequeña, mediana y grande . . . 70
D.16 Traza temporal de fallos en cache para `dedup` con entrada nativa . . . 71
D.17 Fallos por cada mil instrucciones en la cache de datos para `facesim` ejecutado en Intel . . . 72
D.18 Fallos por cada mil instrucciones en la cache de datos para `facesim` ejecutado en Sparc . . . 73
D.19 Traza temporal de fallos en cache para `facesim` con entrada grande (igual a la pequeña y mediana) . . . 73
D.20 Traza temporal de fallos en cache para `facesim` con entrada nativa . . . 74
D.21 Fallos por cada mil instrucciones en la cache de datos para `ferret` ejecutado en Intel . . . 75
D.22 Fallos por cada mil instrucciones en la cache de datos para `ferret` ejecutado en Sparc . . . 76
D.23 Traza temporal de fallos en cache para `ferret` con entradas pequeña, mediana y grande . . . 77
D.24 Traza temporal de fallos en cache para `ferret` con entrada nativa . . . 78
D.25 Fallos por cada mil instrucciones en la cache de datos para `fluidanimate` ejecutado en Intel . . . 79
D.26 Fallos por cada mil instrucciones en la cache de datos para `fluidanimate` ejecutado en Sparc . . . 79
D.27 Traza temporal de fallos en cache para `fluidanimate` con entradas pequeña, mediana y grande . . . 80
D.28 Traza temporal de fallos en cache para `fluidanimate` con entrada nativa . . . 81
D.29 Fallos por cada mil instrucciones en la cache de datos para `freqmine` ejecutado en Intel . . . 82
D.30 Fallos por cada mil instrucciones en la cache de datos para `freqmine` ejecutado en Sparc . . . 83
D.31 Traza temporal de fallos en cache para `freqmine` con entradas pequeña, mediana y grande . . . 84
D.32 Traza temporal de fallos en cache para `freqmine` con entrada nativa . . . 85
D.33 Fallos por cada mil instrucciones en la cache de datos para `raytrace` ejecutado en Intel . . . 86
D.34 Fallos por cada mil instrucciones en la cache de datos para `raytrace` ejecutado en Sparc . . . 86
D.35 Traza temporal de fallos en cache para `raytrace` con entradas pequeña, mediana y grande . . . 87
D.36 Traza temporal de fallos en cache para `raytrace` con entrada nativa . . . 88
D.37 Fallos por cada mil instrucciones en la cache de datos para `streamcluster` ejecutado en Intel . . . 89

D.38 Fallos por cada mil instrucciones en la cache de datos para `streamcluster` ejecutado en Sparc . . . 90
D.39 Traza temporal de fallos en cache para `streamcluster` con entradas pequeña, mediana y grande . . . 91
D.40 Traza temporal de fallos en cache para `streamcluster` con entrada nativa . . . 92
D.41 Fallos por cada mil instrucciones en la cache de datos para `swaptions` ejecutado en Intel . . . 93
D.42 Fallos por cada mil instrucciones en la cache de datos para `swaptions` ejecutado en Sparc . . . 93
D.43 Traza temporal de fallos en cache para `swaptions` con entradas pequeña, mediana y grande . . . 94
D.44 Traza temporal de fallos en cache para `swaptions` con entrada nativa . . . 95
D.45 Fallos por cada mil instrucciones en la cache de datos para `vips` ejecutado en Intel 96
D.46 Fallos por cada mil instrucciones en la cache de datos para `vips` ejecutado en Sparc 97
D.47 Traza temporal de fallos en cache para `vips` con entradas pequeña, mediana y grande . . . 98
D.48 Fallos por cada mil instrucciones en la cache de datos para `x264` ejecutado en Intel 99
D.49 Fallos por cada mil instrucciones en la cache de datos para `x264` ejecutado en Sparc 100
D.50 Traza temporal de fallos en cache para `x264` con entradas pequeña, mediana y granded . . . 101
D.51 Traza temporal de fallos en cache para `x264` con entrada nativa . . . 102

Índice de tablas

3.1 Visión general de las aplicaciones que componen la suite PARSEC. 9

4.1 Configuración del TLB en un Intel Core 2 Duo 13
4.2 Contadores hardware utilizados para medir los fallos de TLB 14

5.1 Selección de las entradas a utilizar con las aplicaciones de PARSEC para conseguir una ejecución representativa en un tiempo razonable. 25

A.1 Número de horas invertidas en cada una de las tareas del proyecto. 35

B.1 Distribución del tiempo de ejecución en ficheros y funciones dentro del módulo Ruby durante una simulación del benchmark `blackscholes` con Simics y GEMS, con 4 procesadores. 43
B.2 Grafo de llamadas para la función en la pasa más tiempo una simulación con Simics y GEMS, `PerfectSwitch::wakeup`. 44

D.1 Características de las entradas de la aplicación `bodytrack` 103
D.2 Características de las entradas de la aplicación `fluidanimate` 105
D.3 Características de las entradas de la aplicación `raytrace` 106
D.4 Características de las entradas de la aplicación `swaptions` 107

Capítulo 1
Introducción

La simulación es un recurso esencial para explorar el espacio de diseño de nuevas arquitecturas de computadores. Para que los resultados de nuestras simulaciones sean fiables nos interesa simular el sistema completo con suficiente precisión. El problema es que no es posible lograr que un simulador sea detallado y, al mismo tiempo, lo suficientemente eficiente como para ejecutar cargas de trabajo realistas en un tiempo razonable. Es imprescindible encontrar programas representativos que proporcionen conclusiones objetivas en que basar el diseño de nuevos sistemas, pero las limitaciones ya descritas nos llevan a utilizar aplicaciones creadas especialmente para este fin que no son necesariamente representativas. Por lo tanto, aunque los resultados obtenidos sean muy precisos, no siempre nos aportan información valiosa.

En este proyecto se va realizar un estudio en profundidad del propio simulador, buscando cuellos de botella cuya optimización reduzca los tiempos de ejecución. Se caracterizarán también cargas de trabajo en lo relativo al diseño de la jerarquía de memorias cache, un aspecto esencial del diseño de todo sistema mono o multiprocesador. Se pretende confirmar o desmentir la creencia popular que indica que cuanto más complejo es el problema a resolver, mayor presión se ejerce sobre la jerarquía de memoria. Además, se buscarán alternativas a la entrada a simular con las cargas de trabajo analizadas, para obtener resultados representativos en el menor tiempo posible. Esto permitirá acelerar el proceso de simulación y garantizará la fiabilidad de los datos en los que se basan las decisiones de diseño.

Como simulador a estudiar nos centraremos en la plataforma Simics (de la empresa Virtutech) [30] y el módulo GEMS (Universidad de Wisconsin) [31]. Simics tiene capacidad para simular un sistema completo (sistema operativo, periféricos, etc.), tanto uniprocesador como multiprocesador, y su uso está actualmente muy extendido. GEMS, por su parte, proporciona módulos para el estudio de prestaciones del sistema de memoria. Como carga de trabajo hemos seleccionado una suite reciente lanzada en 2008 y actualizada en 2009, PARSEC [13], caracterizada por ofrecer, además de un conjunto representativo de las nuevas aplicaciones paralelas emergentes de memoria compartida, una gran variedad en el tamaño del problema a resolver (*input data set*). Esta suite ofrece programas paralelos altamente escalables, lo cual significa que los *threads* paralelos están bastante balanceados [8].

Como resultados principales del proyecto, se presenta el análisis del tiempo de simulación utilizando Simics y GEMS, considerando el efecto de modificar el número de procesadores simulados. Se incluye también un estudio de la variación del *instruction mix* y el número de páginas accedidas (*footprint*) por cada aplicación de PARSEC en función del tamaño de la entrada. Este estudio se ha realizado ejecutando las aplicaciones con un thread, pero los resultados son

extrapolables a cualquier número de threads. Por último, se presenta la selección de entradas o secciones representativas a utilizar para cada aplicación. Esta selección se ha basado en el análisis del comportamiento de las aplicaciones sobre un nivel de memoria cache, ejecutándolas con un thread, aunque se presume que las conclusiones seguirán siendo válidas para múltiples threads. La utilización de esta selección para las aplicaciones de PARSEC resulta más adecuada que el uso sistemático de una entrada de menor tamaño, ya que permite conseguir resultados más fiables manteniendo un tiempo de simulación razonable.

1.1 Contexto del proyecto

Este proyecto se ha desarrollado dentro del Grupo de Arquitectura de Computadores de la Universidad de Zaragoza (gaZ), en relación con el proyecto TIN2010-21291-C02-01 financiado por el Ministerio de Ciencia e Innovación.

Durante el curso 2010/2011 disfruté de una Beca de Colaboración del Ministerio de Educación destinada a la iniciación a la investigación durante la cual me centré en el estudio de las redes de interconexión de caches en multiprocesadores. Actualmente, tengo una beca del Instituto Universitario de Investigación e Ingeniería de Aragón (i3A) que continuará hasta marzo de 2012. Además, durante la realización del proyecto asistí a la escuela de verano internacional ACACES (Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems) gracias a una beca proporcionada por HiPEAC (European Network of Excellence on High Performance and Embedded Architecture and Compilation).

1.2 Objetivos

El objetivo de este Proyecto Fin de Carrera es analizar las plataformas de simulación y establecer cargas de trabajo que faciliten los estudios de diseño de nuevas arquitecturas de computadores. Las tareas de las que consta este proyecto son:

1. Estudio del estado del arte de plataformas de simulación de multiprocesadores y de cargas de trabajo paralelas.
2. Puesta en marcha del entorno de simulación y de las cargas de trabajo.
3. Análisis del entorno de simulación para determinar si hay algún factor responsable de buena parte del tiempo de simulación.
4. Estudio del impacto del tamaño de las entradas de las aplicaciones de PARSEC en la jerarquía de memoria del procesador.
5. Selección de las entradas de las aplicaciones de PARSEC a simular para conseguir resultados representativos en un tiempo razonable.
6. Propuesta de vías de continuación de la investigación.

Con la realización de las tareas anteriormente descritas y como se observa a lo largo de la presente memoria, en particular en los capítulos de resultados y conclusiones del trabajo, se han alcanzado todos los objetivos planteados para este proyecto.

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 de simuladores y cargas de trabajo; en el capítulo 3 se explica con mayor detalle la suite PARSEC; 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 recoge las conclusiones y líneas de trabajo futuro.

Se incluyen como anexos:

A. Gestión del proyecto. Incluye la planificación del tiempo durante el proyecto y el esfuerzo invertido en el mismo.

B. Análisis del tiempo de simulación: Simics y GEMS. Recoge el estudio del tiempo de ejecución de las simulaciones y la distribución de dicho tiempo en los módulos del simulador.

C. Detalles de las simulaciones con Simics y GEMS. Se explica con mayor detalle el proceso seguido para llevar a cabo las simulaciones.

D. Resultados de la caracterización de PARSEC. Se presentan los resultados del impacto del tamaño de la entrada sobre la jerarquía de memoria del procesador, describiendo también el proceso seguido para llevar a cabo la selección de entradas a utilizar para lograr una ejecución representativa en poco tiempo.

Capítulo 2
Estado del arte

En este capítulo se va a realizar una revisión de las plataformas y estrategias utilizadas para la simulación de procesadores. Se comentarán también cuáles son las cargas de trabajo más comúnmente utilizadas.

2.1 Plataformas y estrategias de simulación

Para que los resultados de nuestras simulaciones sean fieles a la realidad se deben ejecutar cargas de trabajo realistas en máquinas simuladas con suficiente detalle. SimpleScalar [6] ha sido un simulador muy utilizado en investigación, pero únicamente puede ejecutar aplicaciones de usuario con un solo *thread*. Además, no ejecuta el código del sistema operativo, lo cual es esencial para programas más complicados. Muchos investigadores están interesados en sistemas que ejecuten cargas de trabajo más complejas, como bases de datos, servidores web y algoritmos científicos paralelos. Por lo tanto, necesitaremos simuladores de sistema completo, que incluyen procesadores, memoria, interfaces de red y otros periféricos. La simulación de sistema completo permite el diseño, desarrollo y prueba de hardware y software en un entorno que se aproxima al contexto final de aplicación del producto.

Virtutech Simics [30] (comúnmente llamado simplemente Simics) es un simulador de sistema completo que podemos configurar para modelar multiprocesadores, sistemas empotrados, routers de telecomunicaciones, clusters o redes de esos elementos. Es capaz de ejecutar sistemas operativos sin necesidad de que sean adaptados y simular aplicaciones realistas ofreciendo resultados precisos. Se trata de un simulador comercial y el código no es libre. Simics suele utilizarse conjuntamente con GEMS (General Execution-Driven Multiprocessor Simulator) [31], que fue creado en la Universidad de Wisconsin y proporciona módulos para el estudio de prestaciones del sistema de memoria y microprocesadores. GEMS está compuesto por Ruby, que simula las caches, el protocolo de coherencia y la red de interconexión, y Opal, para la ejecución fuera de orden. 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 encargará de gestionar los accesos a memoria. Además, en la Universidad de Princeton elaboraron GARNET [4], que, integrado con Ruby, simula detalladamente la red de interconexión en chip. El uso de estas herramientas está muy extendido y son las que se han utilizado a lo largo de este proyecto.

M5 [14] es también un simulador de sistema completo que ha sido adoptado por varios grupos de investigación, tanto en el ámbito académico como en el comercial, gracias a su utilidad como simulador de arquitecturas de proposito general y su licencia de código libre. Recientemente, los creadores de GEMS y M5 iniciaron un proyecto para unir ambas herramientas y crear gem5 [2], que fue presentado en la conferencia ISCA en junio de 2011 y también es de código libre.

Un problema importante en la simulación de multiprocesadores es el bajo rendimiento de los simuladores, que hace que la ejecución de las aplicaciones tarde entre 100 y 100000 veces más en un simulador que en nativo. Todos los simuladores descritos hasta el momento deben ejecutar en serie la simulación de varios procesadores que trabajan en paralelo. Graphite [32] surgió en el MIT como una solución a este problema, presentándose como un simulador paralelo y distribuido que ofrece mejor rendimiento a cambio de sacrificar precisión en los resultados. En la misma línea propusieron también HORNET [29], un simulador paralelo de multiprocesadores que da gran importancia al modelado y rendimiento de la red de interconexión.

Otra opción para acelerar el proceso de simulación es idear nuevas estrategias como, por ejemplo, reducir la cantidad de código del programa que se debe simular. Siguiendo esta idea, en la Universidad de Carnegie Mellon propusieron el método Sampling Microarchitecture Simulation (SMARTS) [41] para obtener medidas del rendimiento de aplicaciones completas de manera rápida y precisa. SMARTS acelera la simulación midiendo en detalle únicamente algunas secciones de la aplicación, que son escogidas mediante muestreo estadístico para obtener el grado de confianza deseado en los resultados. Ekman *et al.* consiguen disminuir el número de puntos a simular en un orden de magnitud manteniendo la precisión que nos interesa aplicando el método estadístico *matched-pair comparison* [20].

2.2 Cargas de trabajo

La selección de las cargas de trabajo que utilizaremos para estudiar el rendimiento de los sistemas simulados tiene también gran importancia [15]. Un *benchmark* es una carga de trabajo artificial que incluye las características más importantes de cargas de trabajo reales y relevantes. Generalmente, los benchmarks son aplicaciones pequeñas, eficientes y controlables.

Los benchmarks de SPEC (Standard Performance Evaluation Corporation) son muy utilizados para la investigación de nuevas arquitecturas. SPEC OMP [36] fue su primera suite creada para la evaluación de prestaciones de memoria compartida basada en OpenMP, dentro del dominio de la computación de altas prestaciones. CPU2006 [37] es parte de la siguiente generación de benchmarks de SPEC, y pretende ser intensiva en cálculo y presionar la jerarquía de memoria, el procesador y el compilador. También tiene como finalidad servir para la comparación de prestaciones entre sistemas distintos.

Splash-2 [39] es un conjunto de benchmarks de 1995 que contiene varias aplicaciones paralelas relacionadas con computación de altas prestaciones y gráficos. Cuando se creó la suite, las plataformas paralelas eran sistemas con varios nodos en los que la comunicación entre nodos era muy costosa. Por ello, los algoritmos intentan minimizar la comunicación entre threads lo máximo posible. La suite es muy popular, aunque los algoritmos se han quedado anticuados para la evaluación de nuevos diseños debido a la proliferación de los multiprocesadores en chip.

EEMBC (The Embedded Microprocessor Benchmark Consortium, pronunciado *embassy*) ha desarrollado varios benchmarks entre los que se encuentran CoreMark y MultiBench. CoreMark [18] es un benchmark simple diseñado específicamente para probar la funcionalidad de un procesador que permite realizar comparaciones rápidamente entre diferentes plataformas. Pero los procesadores son cada vez más complejos y un benchmark destinado a evaluar un solo procesador no es suficiente para realizar un análisis exhaustivo. Más adecuado para este propósito

es MultiBench [19], un conjunto de benchmarks comercial que permite a los diseñadores de sistemas analizar, probar y mejorar plataformas y arquitecturas multicore. MultiBench utiliza cargas de trabajo estandarizadas y es compatible con una amplia variedad de multiprocesadores empotrados y sistemas operativos.

Recientemente, Iqbal *et al.* presentaron ParMiBench [27], que está compuesto por la implementación paralela de siete algoritmos intensivos en cálculo que provienen de la *benchmark suite* para uniprocesadores MiBench [22]. Las aplicaciones pertenecen a cuatro ámbitos distintos: automatización y control industrial, automatización de procesos de oficina, redes y seguridad.

La suite PARSEC (Princeton Application Repository for Shared-Memory Computers) [11, 13, 10, 7, 8, 12] fue creada en Princeton en colaboración con Intel para el diseño de una nueva generación de procesadores. La suite está compuesta por trece aplicaciones multithread representativas de programas emergentes de memoria compartida para multiprocesadores en chip (CMPs). Se ha seleccionado PARSEC como carga de trabajo a utilizar durante el proyecto, por lo que explicamos sus características con más detalle en el capítulo 3.

Capítulo 3
La suite PARSEC

Durante este proyecto nos centraremos en PARSEC, ya introducido en el capítulo 2, por ser un conjunto de benchmarks ampliamente utilizado, precompilado para varias plataformas y ya preparado para ser simulado con Simics. Buena parte de las publicaciones más relevantes en el diseño de multiprocesadores utilizan PARSEC como referencia. En la tabla 3.1 aparecen las trece aplicaciones que componen la benchmark suite junto a una breve descripción.

Aplicación	**Descripción**
`blackscholes`	Cálculos financieros utilizando la ecuación diferencial parcial Black-Scholes.
`bodytrack`	Visión por computador, detección y seguimiento de una persona.
`canneal`	Optimización del coste de enrutamiento en el diseño de un chip.
`dedup`	Compresión de datos usando deduplicación.
`facesim`	Simulación del movimiento de un rostro humano para animación.
`ferret`	Buscador de imágenes por similitud.
`fluidanimate`	Simulación física de fluidos para animación.
`freqmine`	Minería de datos.
`raytrace`	Aplica el algoritmo *raytrace* para animación en tiempo real.
`streamcluster`	Resuelve el problema de *online clustering*.
`swaptions`	Calcula los precios de una cartera de valores usando el modelo Heath–Jarrow–Morton.
`vips`	Procesado de imágenes.
`x264`	Codificación de vídeo en H.264.

Tabla 3.1: Visión general de las aplicaciones que componen la suite PARSEC.

Para abordar el problema del elevado coste en tiempo de las simulaciones, comenzamos analizando el tiempo de simulación de varias aplicaciones de PARSEC con Simics y GEMS. Se observó que la simulación temporal detallada de varios programas con ocho procesadores resultaba más de 1000 veces más lenta que la ejecución de las aplicaciones en nativo, y que este valor seguiría aumentando según incrementáramos el número de procesadores. Estudiando

en mayor profundidad los módulos en los que la simulación invierte más tiempo, se descubrió que gran parte de este tiempo corresponde al módulo Ruby de GEMS. Sin embargo, dentro del módulo la distribución del tiempo es muy dispersa, dificultando en gran medida la optimización del simulador. El estudio y conclusiones detalladas pueden consultarse en el anexo B.

Como la optimización del simulador no es algo trivial si no deseamos comprometer la precisión de los resultados, continuamos estudiando las cargas de trabajo. Los desarrolladores de los benchmarks proporcionan entradas de varios tamaños para utilizar con sus aplicaciones. Según propusieron KleinOsowski *et al.* [28], nos interesarán las siguientes entradas:

a. Una muy pequeña para comprobar el correcto funcionamiento del simulador y realizar pequeñas pruebas, que tarde en ser simulada unos pocos minutos.

b. Otra mayor que nos permita obtener resultados preliminares de rendimiento.

c. Por último, una entrada más realista que nos permita obtener estadísticas de rendimiento reales para la arquitectura que estemos analizando.

La ejecución de las aplicaciones con las dos primeras entradas no es necesariamente representativa de la ejecución con una entrada original completa. Por otro lado, hay una clara necesidad de reducir el tiempo de simulación para una entrada realista, que puede tardar desde unos pocos días hasta varias semanas o meses. La solución pasa por encontrar una manera de reducir los conjuntos de datos de entrada y, en consecuencia, los tiempos de ejecución, manteniendo su representatividad.

En concreto, PARSEC proporciona seis entradas de distintos tamaños:

- `test` Una entrada muy pequeña para probar la funcionalidad básica del programa.
- `simdev` Entrada muy pequeña que garantiza un comportamiento del programa similar al real, destinada a la prueba y desarrollo del simulador.
- `simsmall, simmedium y simlarge` Entradas de diferentes tamaños (pequeña, mediana y grande) adecuadas para el estudio de microarquitecturas con simuladores.
- `native` Entrada muy grande destinada a la ejecución nativa. Consideraremos que se trata de una aproximación a la ejecución con una entrada realista.

Cuando este benchmark se utiliza en investigación, las aplicaciones se simulan con las entradas pequeña, mediana o grande (por ejemplo, [23, 21, 9, 38, 33] entre muchas otras). Biena *et al.* analizaron el escalado de las entradas en [10], considerando que las entradas para simulación deberían ser aproximaciones de la entrada nativa y ser capaces de proporcionar resultados significativos. Pero en su estudio no compararon las entradas pequeña, mediana y grande con la nativa, alegando que esta última tiene un valor práctico muy limitado ya que es inviable utilizarla en simulaciones. Por lo tanto, no está demostrado que las entradas destinadas a simulación sean adecuadas para obtener resultados representativos y se utilizan simplemente porque el tiempo de simulación resulta conveniente.

En este proyecto, proponemos realizar un estudio de las diferentes entradas para cada programa, incluyendo la nativa, para encontrar de qué manera se pueden lograr unos resultados representativos de la ejecución real con un tiempo de simulación aceptable.

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 Introducción a las métricas utilizadas

Para comenzar el estudio de la suite PARSEC es necesario seleccionar las métricas que se utilizarán para caracterizar el comportamiento de los programas y establecer conclusiones. Se listan a continuación las estadísticas que se decidió recoger:

- El *instruction mix* es el número de instrucciones de cada tipo que hay en un programa, ya sean aritmético-lógicas, de memoria,... Dentro de las de memoria podemos examinar la proporción de operaciones de lectura y escritura que se ejecutan con cada una de las entradas de una aplicación para comprobar si la relación se ha mantenido al realizar el escalado.
- El *footprint* (huella) es el número total de páginas a las que un programa accede cuando es ejecutado. Nos servirá para ver cuánta memoria utiliza cada una de nuestras aplicaciones y las diferencias existentes entre las entradas.
- El TLB (*Translation Lookaside Buffer*) es una tabla utilizada en sistemas de memoria virtual que almacena la dirección física asociada a la dirección virtual de la página para acelerar el proceso de traducción. Los fallos de TLB pueden tener en muchos casos gran impacto en el rendimiento del sistema.
- La tasa de fallos en cache, variando la capacidad de la misma. Frecuentemente, la tasa de fallos no va decreciendo de forma continua al aumentar el tamaño de la cache, sino que se mantiene en un cierto nivel y después baja bruscamente a otro inferior cuando la capacidad es suficientemente grande como para que quepa la siguiente estructura de datos importante.
- Una traza temporal del comportamiento de cada aplicación en la cache, que nos permitirá ver la aparición de los fallos a lo largo del tiempo y así detectar posibles patrones repetitivos y comparar las diferentes entradas con mayor detalle.

Para todas las métricas nos interesará tener en cuenta únicamente la región de interés del programa (*region of interest* o ROI), que es la parte que se ejecutará en paralelo al utilizar varios threads. Es decir, eliminamos del análisis aquellas partes en las que estamos cargando los datos que va a utilizar nuestra aplicación y en las que se escribe el resultado final.

Aunque PARSEC es una benchmark suite destinada a estudiar el comportamiento de multiprocesadores, todo el estudio se ha llevado a cabo ejecutando las aplicaciones con un solo thread, lo cual nos permite analizar el comportamiento de los benchmarks y comparar las diferencias entre las entradas. Los resultados obtenidos para el instruction mix y el footprint serán válidos al aumentar el número de threads de la aplicación, a pesar de que es posible que se ejecuten nuevas instrucciones correspondientes a la sincronización de threads que alteren ligeramente el instruction mix y que al utilizar más threads se necesite asignar espacio dinámicamente para alguna estructura de datos adicional que aumente el footprint.

También suponemos que las conclusiones obtenidas a partir de las tasas de fallos y trazas temporales serán extrapolables a un entorno multiprocesador, aunque queda como trabajo futuro confirmar esta hipótesis. Un estudio más amplio queda fuera del alcance de este proyecto fin de carrera debido a la limitación temporal y la complejidad añadida al introducir variaciones en el número de threads y procesadores.

4.2 Footprint de la memoria

Para calcular el footprint hemos partido de una herramienta desarrollada dentro del gaZ por Alastruey *et al.* [5]. Esta herramienta utiliza SHADE [35], un emulador de hardware SPARC, para reconocer los accesos a memoria y usa esta información para ir almacenando el número de veces que el programa accede a cada bloque de datos, siendo el tamaño de bloque configurable. Al final de la ejecución obtenemos el footprint de la memoria para instrucciones y datos, diferenciando si los accesos son de lectura o de escritura. Además, se ordenan los bloques en orden descendente según el número de referencias a cada uno de ellos y se selecciona el menor número de bloques posibles que acumulen un porcentaje de accesos que nos interese. De esta forma conseguimos un footprint de la memoria del 50 % o el 90 %, lo cual nos servirá para tener una idea de la localidad que presentan las aplicaciones.

Por ejemplo, en la figura 4.1 se muestra el footprint del 50 %, 90 % y 100 % para todas las entradas de la aplicación `blackscholes`. Nótese que la escala es logarítmica para facilitar la representación de los datos. Se ve que, en todos los casos, el 50 % de los accesos a memoria caen sobre únicamente 8 KB, y el 90 %, sobre unos 32 KB. El footprint total de la aplicación va desde unos 700 KB hasta más de 600 MB dependiendo de la entrada, lo que nos hace concluir que este programa presenta mucha localidad espacial.

En nuestro caso, nos interesaba ejecutar las aplicaciones en un Intel de 64 bits (en concreto, nuestra máquina local es un Intel Core 2 Duo y en el cluster del departamento se dispone de máquinas Intel Xeon), así que no podíamos usar la herramienta directamente. Decidimos integrar la parte correspondiente a las estructuras de datos y los cálculos para obtener las métricas con Pin [26], una herramienta de Intel para la instrumentación dinámica de programas. Por otro lado, nosotros queremos medir únicamente la región de interés del programa y despreciar el código correspondiente a la inicialización y finalización. Para ello, añadimos, mediante opciones de configuración, la posibilidad de comenzar y detener la instrumentación al ejecutar funciones determinadas o en las direcciones de PC que nos interese. Además, previendo que en un futuro nos interesaría analizar los programas variando el número de threads, hemos añadido soporte para programas multithread. Al medir el footprint de las aplicaciones de PARSEC hemos configurado la herramienta con un tamaño de bloque de 4KB para obtener como resultado los accesos a cada página de la memoria.

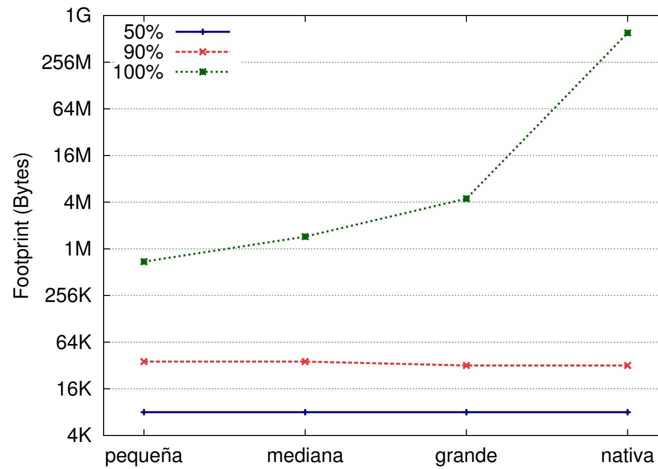


Figura 4.1: 50 %, 90 % y 100 % del footprint de la aplicación `blackscholes`

4.3 Obtención de los fallos de TLB

El ordenador en el que estamos trabajando, un Intel Core 2 Duo, tiene dos niveles de TLB con la configuración que se describe en la tabla 4.1. Las lecturas de datos pasan por el TLB de datos de nivel L0, y si fallan van al nivel superior. En cambio, las escrituras van directamente al nivel superior.

Tipo de TLB	Tamaño de página	Asociatividad	Número de entradas
TLB de datos (L0)	4 KB	4	16
TLB de datos	4 KB	4	256
TLB de instrucciones	4 KB	4	128

Tabla 4.1: Configuración del TLB en un Intel Core 2 Duo

Para obtener los fallos de TLB hemos utilizado VTune [25], una herramienta de *profiling* para el estudio del comportamiento de un programa que resulta muy útil para la optimización del rendimiento. Para monitorizar el rendimiento del hardware, VTune utiliza los contadores hardware del procesador y muestreo basado en eventos (*event based sampling* o EBS). Este método se basa en interrumpir la aplicación cada cierto número de eventos y anotar en qué punto del código se encuentra. De esta manera se obtiene un histograma del número eventos basado en las líneas de código en que se producen. Como se utiliza este método de muestreo estadístico, hemos tomado como resultados finales la media de los valores obtenidos en diez ejecuciones.

En este caso hemos tomado las estadísticas de la ejecución completa de la aplicación, no únicamente de la región de interés, ya que habría sido necesario incorporar instrucciones especiales al código y recompilar las aplicaciones, lo que habría complicado mucho el proceso. De todas formas, la región de interés supone la mayor parte del tiempo total de ejecución de las aplicaciones.

Todos los contadores de eventos hardware disponibles para los procesadores Intel pueden consultarse en [24]. En concreto, nosotros hemos utilizado los contadores descritos en la tabla 4.2 para monitorizar el rendimiento del TLB.

Contador	**Descripción**
DTLB_MISSES.L0_MISS_LD	Cuenta el número de fallos en el TLB de datos de nivel 0 debidos a instrucciones `load`. Incluye fallos detectados como resultado de accesos especulativos.
DTLB_MISSES.ANY	Cuenta el número de fallos en el TLB de datos. Incluye fallos detectados como resultado de accesos especulativos.
DTLB_MISSES.MISS_LD	Cuenta el número de fallos en el TLB de datos debidos a instrucciones `load`. Incluye fallos detectados como resultado de accesos especulativos.
DTLB_MISSES.MISS_ST	Cuenta el número de fallos en el TLB de datos debidos a instrucciones `store`. Incluye fallos detectados como resultado de accesos especulativos.

Tabla 4.2: Contadores hardware utilizados para medir los fallos de TLB

4.4 Instrumentación del programa utilizando VALGRIND

VALGRIND [3] es un sistema de instrumentación que proporciona algunas herramientas para el depurado y profiling de programas y permite construir otras. En concreto, Cachegrind realiza una simulación detallada de las caches I1 (cache de instrucciones de primer nivel), D1 (cache de datos de primer nivel) y L2 (cache compartida de segundo nivel), devolviéndonos el número de accesos a memoria, fallos de cache e instrucciones ejecutadas para cada línea de código. Nosotros hemos utilizado Callgrind, una extensión de la herramienta anterior que nos proporciona también información relativa al grafo de llamadas y algunas opciones extra de instrumentación.

Para obtener estadísticas únicamente de la región de interés, utilizamos las opciones de configuración de VALGRIND para indicarle que ponga a cero todos los contadores justo antes de comenzar la zona de código que nos interesa y que escriba las estadísticas al terminarla.

Hemos usado VALGRIND para obtener el *instruction mix*, ya que nos proporciona información del número de operaciones de lectura y escritura que ejecuta el programa.

También queremos analizar el número de fallos de lectura y escritura que se producen en la cache de datos. Además, es interesante observar cómo va variando el número de fallos cuando incrementamos la capacidad de la cache. Este estudio suele hacerse con un único nivel de cache, pero como VALGRIND nos obliga a utilizar necesariamente dos niveles, hemos ido variando el tamaño de la cache D1 y hemos mantenido una L2 grande siempre del mismo tamaño. En concreto, se han analizado caches cuyo tamaño crece exponencialmente desde 4 KB hasta 32 MB, con asociatividad 8 y tamaño de bloque de 64 B.

4.5 Estudio de la jerarquía de memoria mediante simulación

Para el estudio de la jerarquía de memoria se han realizado varias simulaciones utilizando Simics 3.0.31, ya introducido en el capítulo 2. En el anexo C se describe con mayor detalle la metodología seguida para llevar a cabo las simulaciones. El uso de este simulador nos permite ejecutar los programas de manera controlada en arquitecturas con diferentes configuraciones. Nosotros simularemos una arquitectura UltraSPARC III Plus en la que ejecutaremos el sistema operativo Solaris 10, partiendo de una configuración ya elaborada por Jorge Albericio dentro del gaZ.

Una gran ventaja de utilizar un simulador es la posibilidad de crear *checkpoints*, que permiten guardar toda la configuración del punto de la simulación en el que nos encontramos para volver a él rápidamente más tarde. Por ejemplo, lo primero que será necesario hacer es iniciar el sistema operativo, pero podemos guardar un checkpoint al terminar y comenzar a partir de ahí el resto de las veces.

Para ejecutar los benchmarks en el sistema simulado necesitaremos primero copiar todos los ficheros desde nuestro ordenador. Simics nos permite copiar toda la información que deseemos montando en el sistema simulado, *target*, una carpeta correspondiente a la máquina en la que estamos simulando, *host*. Además, podemos comenzar a ejecutar las aplicaciones y crear un checkpoint antes de iniciar la región de interés, de manera que no tengamos que volver a simular el código que precede a la ROI el resto de las veces.

Decidimos utilizar el simulador para analizar también los fallos de lectura y escritura variando la capacidad de la cache. En el estudio presentado en el anexo B se había utilizado el módulo Gems para simular la jerarquía de memoria, pero en este caso no necesitábamos mucho detalle en los resultados. Por lo tanto, optamos por utilizar el sistema de caches proporcionado por Simics, que no ralentiza la simulación tanto como Gems. Además, a diferencia de VALGRIND, Simics sí que nos permite utilizar únicamente una cache para datos y obviar el resto de la jerarquía de memoria. Al igual que antes, simulamos una cache con tamaño desde 4 KB a 32 MB, asociatividad 8 y tamaño de bloque de 64 B.

Para que las estadísticas recogidas correspondan sólo al código de usuario y no a instrucciones del sistema, hemos añadido una función que se ejecuta cada vez que se cambia entre modo de usuario y modo protegido y se encarga de activar y desactivar la cache para que las instrucciones de sistema no pasen por ella.

Simics también ofrece la posibilidad de detener la ejecución tras un número concreto de instrucciones o ciclos, lo que nos ha permitido obtener la traza temporal de los fallos en cache. Hemos realizado estas simulaciones con una cache de 64 KB. Para las entradas pequeña, mediana y grande se han obtenido las estadísticas cada diez millones de instrucciones. Para la entrada nativa resultaba demasiado costoso en tiempo tomar las estadísticas tantas veces. Se consideró aumentar el intervalo a cien millones de instrucciones, pero para poder comparar estos resultados con los anteriores era importante mantenerlo constante durante todas las pruebas. Finalmente, se decidió tomar las estadísticas cada diez millones de instrucciones, pero no durante toda la ejecución completa, sino en diez intervalos de tamaño igual a la entrada grande escogidos al azar. De este modo podíamos obtener una muestra representativa y fácilmente analizable de la totalidad de la ejecución, pero reduciendo notablemente el tiempo de simulación.

Para acelerar los periodos de simulación correspondientes a intervalos en los que no tomamos medidas, dejamos de utilizar la cache y la función que se ejecuta al cambiar entre modo de usuario y modo protegido. El problema de utilizar este método es que las estadísticas correspondientes al inicio del intervalo no son válidas porque la cache todavía no contiene ningún bloque (en [40] hacen referencia a este problema y proponen una solución para el cálculo de la tasa de fallos). Por ello, para que todos los resultados recogidos durante el intervalo sean válidos, calentaremos las caches durante cien millones de ciclos antes de comenzar de manera que contengan datos como si hubieran estado utilizándose durante toda la ejecución de la aplicación.

Capítulo 5
Resumen de resultados

Este capítulo presenta las principales ideas obtenidas a partir del análisis del tiempo de simulación y el impacto del tamaño de las entradas en la jerarquía de memoria. Se incluye también un resumen de la selección de las entradas que proponemos para obtener resultados representativos en el menor tiempo posible. En los anexos B y D se describen con mayor detalle los resultados y el razonamiento seguido para la selección de entradas.

5.1 Análisis del tiempo de simulación

Recordamos que un gran obstáculo para las simulaciones es que resultan muy costosas en tiempo. Como una primera aproximación al problema, se realizó un análisis del tiempo de simulación cuyos detalles y conclusiones pueden consultarse en el anexo B.

Primero, analizamos cuánto se ralentizaba la ejecución de las aplicaciones de PARSEC que presentan una mayor tasa de fallos en cache (según [8]) al simularlas con Simics y GEMS. Las simulaciones funcionales, usando sólo Simics, con uno y dos procesadores resultan entre 1.5 y 5 veces más lentas que la ejecución de las aplicaciones en nativo. Cuando añadimos el módulo de GEMS para realizar simulación temporal, teniendo en cuenta los detalles de la jerarquía de memoria y la coherencia, la simulación con un procesador llega a ser hasta 177 veces más lenta que la ejecución nativa, y con dos procesadores este valor llega hasta 385. Esta cifra sigue aumentando a medida que incrementamos el número de procesadores, alcanzando valores superiores a 1000 con ocho procesadores.

A continuación, se estudió en mayor profundidad en qué módulos del simulador se invierte más tiempo de ejecución esperando encontrar un cuello de botella que pudiéramos optimizar para lograr una mejora en el rendimiento. Utilizamos tres aplicaciones con diferentes características en cuanto al ámbito de aplicación, tipo y granularidad del paralelismo y tamaño del *working set*, y simulamos dos, cuatro y ocho procesadores. Los resultados muestran que, al simular la jerarquía de memoria completa, gran parte del tiempo de la simulación corresponde a la ejecución del módulo Ruby de GEMS, pero dentro de este módulo la distribución del tiempo es muy dispersa, eliminando la posibilidad de que una mejora en una zona específica de código tenga un impacto relevante en la ejecución global.

5.2 Impacto del tamaño de las entradas en la jerarquía de memoria

En esta sección sintetizamos los resultados obtenidos a partir de los experimentos destinados a analizar el impacto del tamaño de las entradas en la jerarquía de memoria descritos en el capítulo 4.

5.2.1 Instruction mix

En la figura 5.1 se muestra el porcentaje de cada tipo de instrucción (lectura, escritura y otras) que se ejecuta en las aplicaciones de la suite PARSEC con cada una de las entradas. Claramente podemos ver que la proporción de instrucciones de lectura y escritura respecto del total se mantiene aunque aumentemos el tamaño de la entrada. Por otro lado, el número de instrucciones sí que es significativamente mayor cuanto más grande es la entrada (los valores concretos pueden consultarse la figura D.1). Esto nos indica que, efectivamente, las entradas más pequeñas son una aproximación reducida de las más grandes.

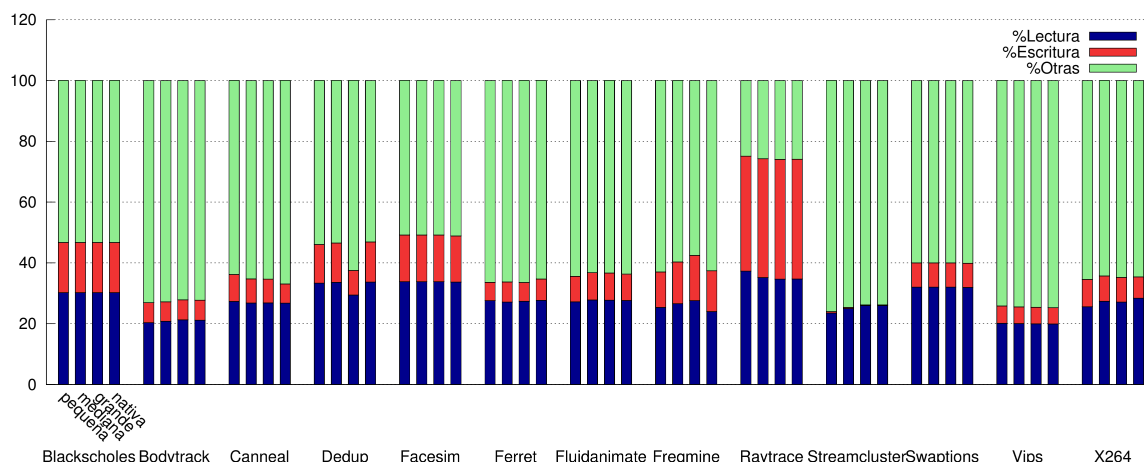


Figura 5.1: Instruction mix de las aplicaciones de PARSEC con sus entradas pequeña, mediana, grande y nativa.

5.2.2 Footprint

Representamos el footprint en la figura 5.2, en la que aparece el número de bytes utilizado en la ejecución de todas las aplicaciones con sus entradas (recordamos que cada página tiene un tamaño de 4 KBytes). Diferenciamos además los bytes que corresponden a datos del programa y los que almacenan las instrucciones ejecutadas. El tamaño de la memoria que utiliza cada aplicación para instrucciones se mantiene constante, pero la que se usa para datos aumenta con el tamaño de la entrada, lo que nos lleva a pensar que las entradas mayores realizarán un uso más exhaustivo de la memoria y, por lo tanto, ejercerán más presión sobre la jerarquía.

Por otro lado, se han creado gráficas con el footprint del 50 %, 90 % y 100 %, tal y como se ha explicado en la sección 4.2 (veíamos ya un ejemplo en la figura 4.1 y podemos analizar los resultados para el resto de las aplicaciones en las figuras D.3 y D.4). Vemos que todas las aplicaciones presentan mucha localidad, ya que la mayor parte de los accesos se concentran en

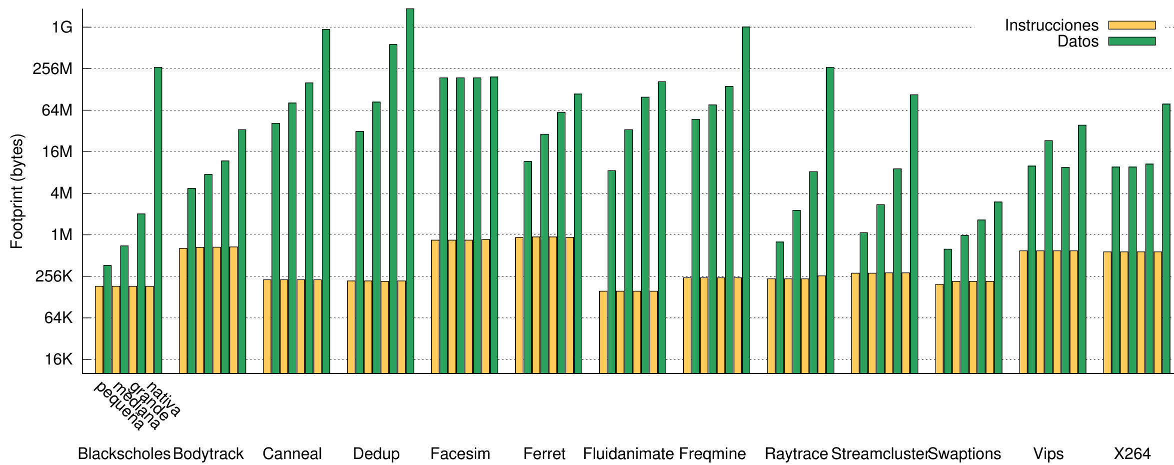


Figura 5.2: Footprint de las aplicaciones de PARSEC con sus entradas pequeña, mediana, grande y nativa.

un conjunto pequeño de las páginas de memoria.

5.2.3 Fallos de TLB

En la figura 5.3 representamos en número de fallos por cada mil instrucciones que generan las operaciones de lectura y escritura en los dos niveles del TLB. Estos fallos no siguen ya el mismo patrón que se veía en el instruction mix y en el footprint. El número de fallos no es mayor con las entradas de mayor tamaño, sino que va disminuyendo según pasamos a entradas más grandes (`blackscholes`, `bodytrack`, `ferret`, `raytrace` y `vips`) o crea una forma de "U", es decir, las entradas pequeña y nativa presentan más fallos que la mediana y grande (`canneal`, `dedup`, `fluidanimate`, `swaptions` y `x264`). Estos resultados nos indican que no es correcto suponer que las entradas nativas presentarán más fallos y que cuanto más grande sea la entrada más nos acercaremos a una ejecución real.

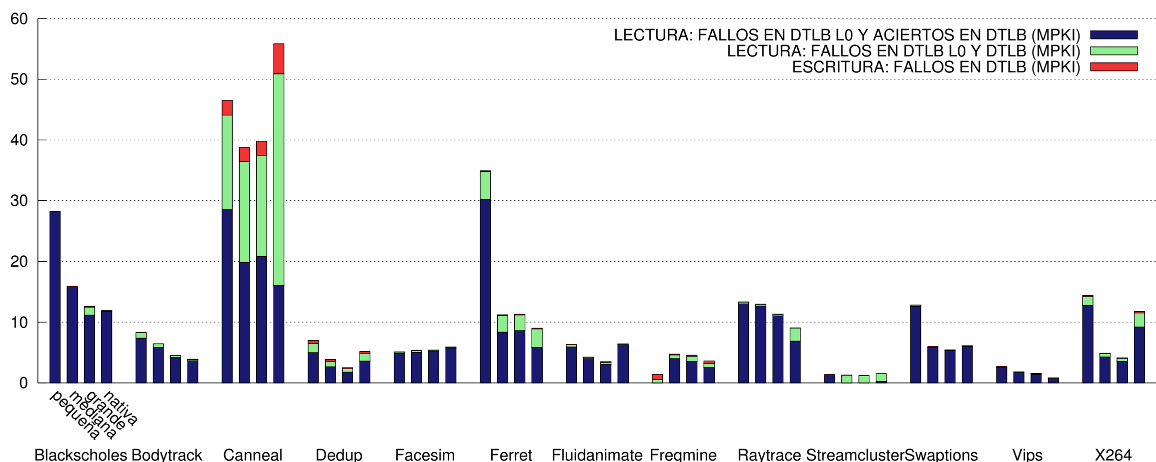


Figura 5.3: Fallos por cada mil instrucciones en el TLB de datos de las aplicaciones de PARSEC con sus entradas pequeña, mediana grande y nativa. Se muestran los fallos que se producen para las lecturas que acceden al nivel L0 (DTLB L0), y los fallos de lectura y escritura del nivel superior (DTLB)

5.2.4 Tasa de fallos en cache y traza temporal

Esta sección recoge los fallos en cache por cada mil instrucciones (MPKI) para las diferentes entradas variando la capacidad de la cache, con arquitectura Intel y Sparc, y las trazas temporales de fallos. Presentaremos en detalle los resultados para la aplicación `blackscholes` y el resto podrán consultarse en el anexo D.

En las figuras 5.4 y 5.5 representamos las tasas de fallos en cache para una arquitectura Intel, utilizando una política *write-allocate* y *copy-back*, y para una arquitectura Sparc, tanto con política write-allocate y copy-back como con *non-write-allocate* y *write-through*. En el eje x (horizontal) aparecen todos los tamaños de cache utilizados y, para cada uno de ellos, las cuatro entradas de la aplicación. En el eje y (vertical) representamos el número de fallos en lectura y escritura expresados en MPKI. En todos los casos se ve cómo, según aumenta la capacidad de la cache, el número de fallos para las entradas más pequeñas va disminuyendo drásticamente a partir del punto en que las estructuras principales caben en la cache. Por lo tanto, en este caso la entrada nativa sí que genera más fallos en cache y estresa más la jerarquía de memoria.

Si nos fijamos en las dos gráficas para la arquitectura Sparc (figura 5.5) podemos apreciar claramente que la de política non-write-allocate presenta muchos más fallos en escritura que la write-allocate. Esto se debe a que no traemos nunca a memoria los bloques cuando se produce un fallo en escritura, así que se fallará repetidamente. De todas formas, vemos que la relación entre los fallos de las entradas se mantiene constante sin excepción. Esto se repite en los apartados siguientes y lo tendremos en cuenta para obtener las conclusiones en los casos en que no se dispone de todos los resultados.

Para las trazas temporales presentamos dos gráficas de cada una de las ejecuciones o muestras (figuras 5.6 y 5.7). En la gráfica superior aparece el número absoluto de accesos a memoria en cada intervalo, para lectura y escritura. En la parte inferior, podemos ver el número de fallos por cada mil instrucciones. En el eje x se indica el punto de ejecución de la aplicación en que nos encontramos, representado en el número de intervalos de diez millones de ciclos. Para las entradas pequeña, mediana y grande, la longitud del eje x se corresponde con el número de ciclos del total de la ejecución del programa. Por lo tanto, la ejecución con la entrada pequeña tarda unos 180 millones de ciclos, con la entrada mediana, algo más de 700 millones y con la entrada grande, casi 3000 millones. Para la entrada nativa se representan por separado cada una de las diez muestras tomadas, y mirando los ciclos del eje x se puede ver a qué parte de la ejecución corresponden. Además, el procesador de Simics es muy simple y mantiene un IPC (instrucciones por ciclo) de uno en todo momento, así que en un intervalo de diez millones de ciclos se ejecutarán diez millones de instrucciones, aunque el número de instrucciones de usuario será ligeramente menor.

Analizando las trazas temporales vemos que el número de fallos se mantiene prácticamente constante a lo largo de toda la ejecución, presentado pequeños picos periódicamente.

Las mismas gráficas para el resto de aplicaciones se presentan en las figuras D.5 a D.51, acompañadas de explicaciones detalladas adicionales.

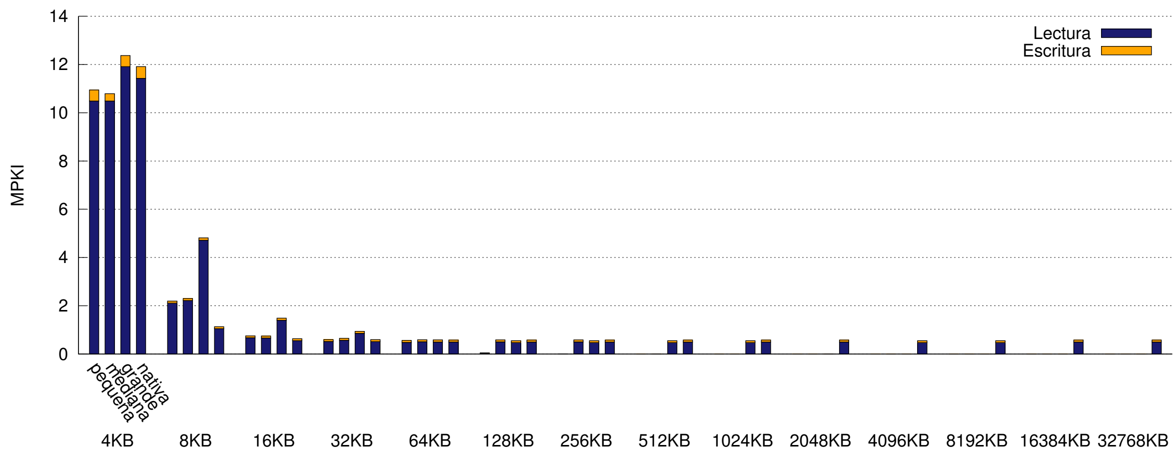
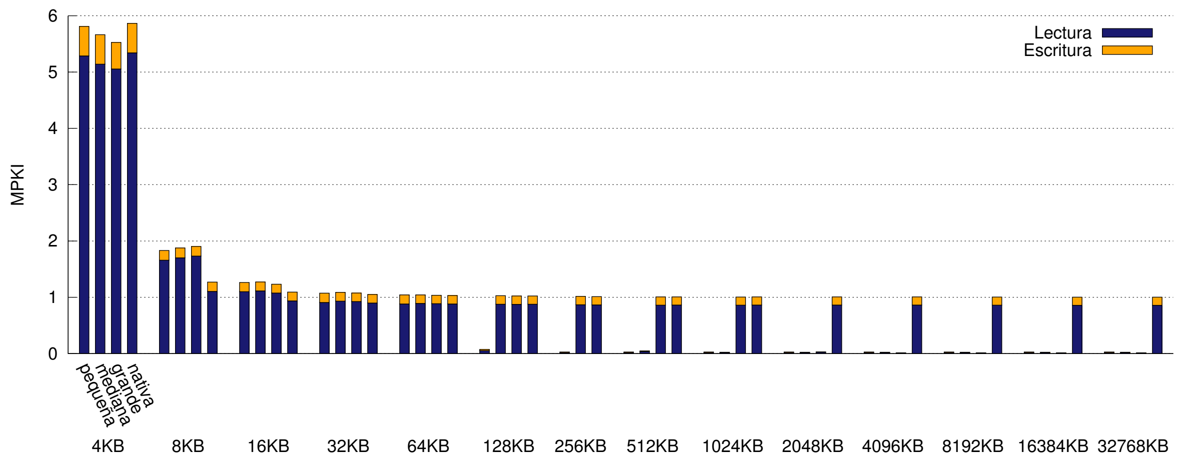
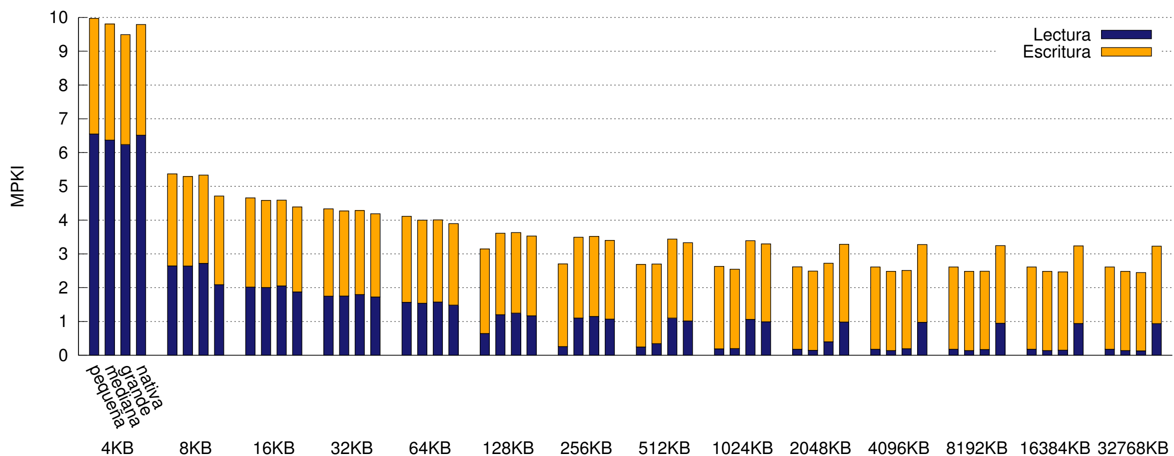


Figura 5.4: Fallos por cada mil instrucciones en la cache de datos para `blackscholes` ejecutado en Intel. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

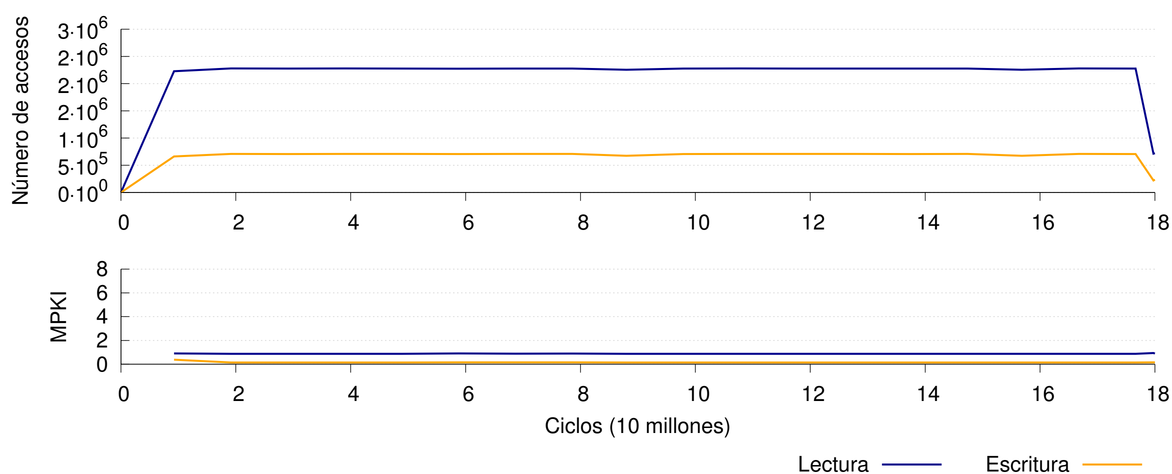


(a) write-allocate y copy-back, sólo instrucciones de usuario

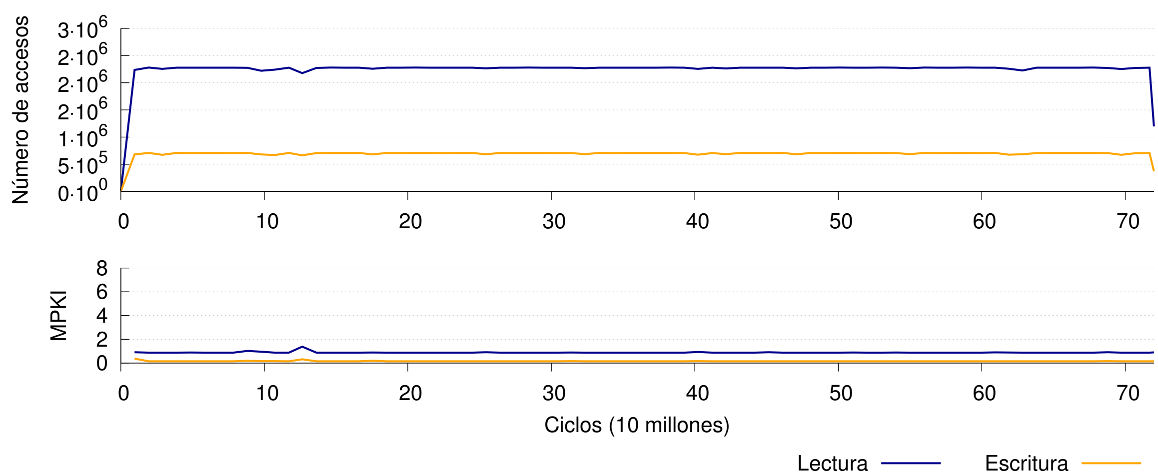


(b) non-write-allocate y write-through, instrucciones de usuario y sistema

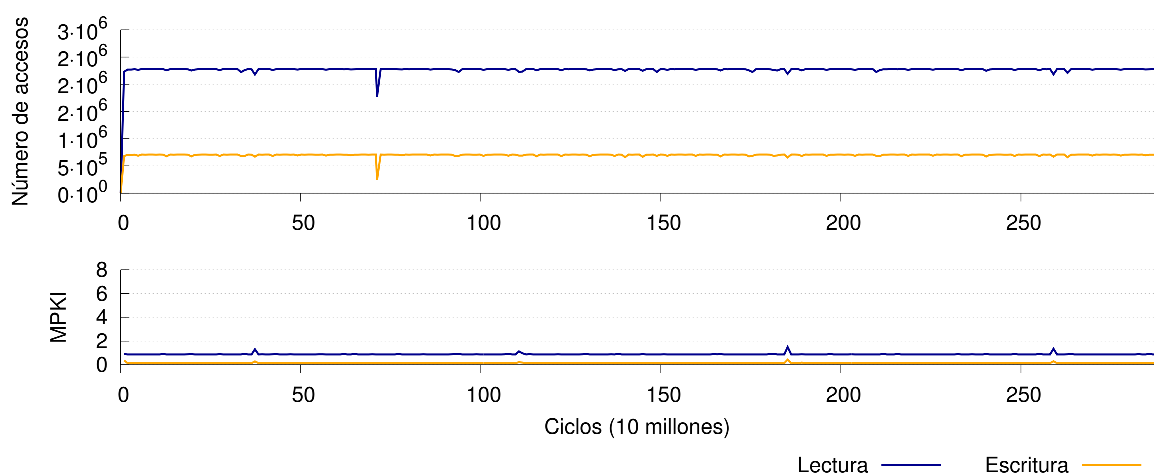
Figura 5.5: Fallos por cada mil instrucciones en la cache de datos para `blackscholes` ejecutado en Sparc.



(a) pequeña



(b) mediana



(c) grande

Figura 5.6: Traza temporal de fallos en cache para `blackscholes` con entradas pequeña, mediana y grande, ejecutado en Sparc. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

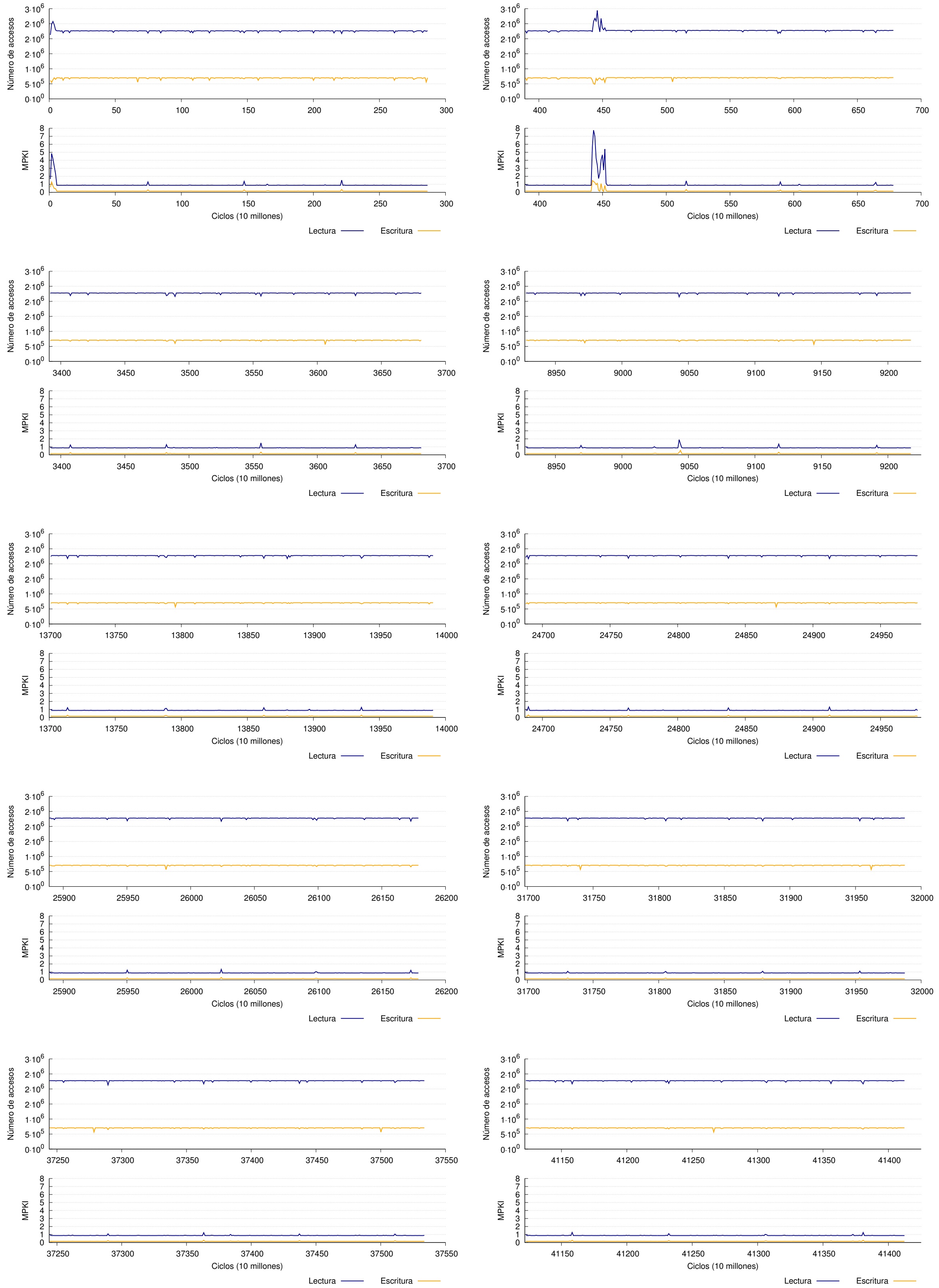


Figura 5.7: Traza temporal de fallos en cache para `blackscholes` con entrada nativa, ejecutado en Sparc. Aparecen diez muestras tomadas al azar del total de la ejecución. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

Principalmente, debemos destacar que no siempre presentan más fallos las entradas de mayor tamaño. En algunos casos se aprecia claramente que al aumentar el tamaño de la cache, los fallos de las entradas de más pequeñas van disminuyendo drásticamente a partir del punto en el que las estructuras de datos principales caben en la cache (`blackscholes`, `fluidanimate` y `streamcluster`). Pero hay otros casos en los que no se aprecia apenas ninguna diferencia entre las cuatro entradas (`facesim` y `swaptions`) o en los que hay entradas de menor tamaño que presentan más fallos que las nativas (`bodytrack`, `ferret`, `freqmine` y `vips`). En cualquier caso, en contra de lo que se pudiera pensar anteriormente, los fallos para las entradas de menor tamaño son perfectamente comparables a los de la entrada nativa. Una aproximación como la de utilizar una cache de tamaño extremadamente pequeño con una entrada pequeña para aproximar el comportamiento de una ejecución nativa ([16]) será totalmente incorrecta. También es importante señalar que algunos de los benchmarks presentan un número de fallos especialmente bajo, haciendo que su uso para el estudio de la jerarquía de memoria sea muy inadecuado (`raytrace` y `swaptions`).

La traza temporal de los fallos en cache nos aporta información muy útil en la mayor parte de los casos. Hay algunos benchmarks en los que los fallos se mantienen estables durante toda la ejecución (`blackscholes`, `canneal`, `streamcluster`, `swaptions` y `vips`), pero hay otros en los que se distingue claramente un patrón que se repite para las cuatro entradas (`bodytrack`, `facesim`, `fluidanimate` y `raytrace`). Esto último nos ha permitido además crear una correspondencia directa entre la forma de la traza y las características de cada entrada, sabiendo qué se está ejecutando en cada momento.

5.3 Selección de entradas

Ante los resultados obtenidos en la sección 5.2.4, resultaba claro que en la mayor parte de los casos se podía conseguir una ejecución representativa de la nativa sin necesidad de simular tantas instrucciones. Por lo tanto, realizamos una selección de las entradas más adecuadas que se deberán utilizar para llevar a cabo un estudio de la jerarquía de memoria. Para realizar la selección utilizamos varias técnicas distintas, según las necesidades de cada aplicación. Las técnicas empleadas son las siguientes:

- Ejecución de una sección de la entrada nativa. Cuando hay diferencias entre las entradas de diferente tamaño pero el número de fallos a lo largo de la ejecución de la entrada nativa se mantiene uniforme, es suficiente con ejecutar una sección de la entrada nativa para obtener resultados representativos de la simulación completa. En este caso habrá que tener en cuenta que, si hay que ejecutar una sección tomada de un punto central o aleatorio de la región de interés, será necesario calentar las caches previamente.

- Uso de una entrada de tamaño menor. Hay casos en los que no hay diferencias entre las entradas o en los que una entrada de menor tamaño resulta más adecuada por presionar más a la jerarquía de memoria.

- Uso de una nueva entrada. En algunas aplicaciones hemos detectado que todas las entradas realizan repeticiones de algún patrón. Las entradas pequeñas tienen menos iteraciones y estructuras de menor tamaño y la entrada nativa hace muchas repeticiones con estructuras mayores. Para conseguir una ejecución representativa de la nativa pero en menor tiempo podemos hacer tantas iteraciones como la entrada de menor tamaño, pero realizarlas con unas estructuras tan grandes como las de la entrada más grande.

En la tabla 5.1 indicamos qué técnica se ha utilizado para la selección de la entrada de cada benchmark y algún comentario adicional que explica brevemente en qué consistiría la ejecución. Para más detalles del proceso seguido para realizar la selección en cada caso se puede consultar el anexo D.

Aplicación	**Técnica**	**Comentario**
`blackscholes`	Sección de la nativa	Ejecutar primeros 750 millones de instrucciones de la ROI.
`bodytrack`	Nueva entrada	Utilizar 1 fotograma con 4000 partículas.
`canneal`	Sección de la nativa	Ejecutar los primeros 1500 millones de instrucciones de la ROI.
`dedup`	Entrada nativa	El pipeline del algoritmo está muy desbalanceado, el escalado de las entradas es muy malo.
`facesim`	Entrada de menor tamaño	Usar la entrada pequeña.
`ferret`	Entrada de menor tamaño	Usar la entrada pequeña para obtener la ejecución más similar a la nativa. Si se desea maximizar el número de fallos en cache, usar la entrada grande.
`fluidanimate`	Nueva entrada	Utilizar 5 fotogramas con 500000 partículas.
`freqmine`	Entrada de menor tamaño	Usar la entrada pequeña, que es la que más fallos en cache presenta.
`raytrace`	Nueva entrada	Usar 3 fotogramas con 10 millones de polígonos y resolución 1920x1080. *Aplicación poco adecuada para el estudio de la jerarquía de memoria porque presenta muy pocos fallos en cache.*
`streamcluster`	Sección de la nativa	Ejecutar 10000 millones de instrucciones saltando los 5000 millones al inicio de la ROI.
`swaptions`	Entrada de menor tamaño	Usar la entrada pequeña. *Aplicación poco adecuada para el estudio de la jerarquía de memoria porque presenta muy pocos fallos en cache.*
`vips`	Entrada de menor tamaño	Usar la entrada pequeña, que es la que más fallos en cache presenta.
`x264`	Sección de la nativa	Tomar cuatro muestras de 20000 millones de instrucciones en puntos aleatorios de la ROI.

Tabla 5.1: Selección de las entradas a utilizar con las aplicaciones de PARSEC para conseguir una ejecución representativa en un tiempo razonable.

Capítulo 6
Conclusiones y trabajo futuro

6.1 Conclusiones a nivel técnico y trabajo futuro

La simulación es imprescindible para el estudio y diseño de nuevas arquitecturas de computadores, suponiendo un problema fundamental su elevado coste en tiempo. Durante este proyecto se ha abordado dicho problema utilizando el simulador Simics [30] con el módulo GEMS [31] y la benchmark suite PARSEC [13].

Comenzamos realizando un estudio del tiempo de simulación de Simics y GEMS, a partir del cual determinamos que el *slowdown* de las simulaciones aumenta linealmente con el número de procesadores simulados, llegando a adoptar valores superiores a 1000 con sólo ocho procesadores. A pesar de que gran parte del tiempo de ejecución se debe al módulo Ruby de GEMS, la distribución del tiempo dentro del mismo es muy dispersa. La inexistencia de un cuello de botella en el simulador dificulta en gran medida la aplicación de una optimización que tenga un efecto suficientemente apreciable en el tiempo total de simulación.

Ya que es inviable optimizar el simulador, se han buscado cargas de trabajo más ligeras que permitan obtener resultados representativos sin suponer tiempos de simulación excesivamente elevados. Se ha analizado el comportamiento de las trece aplicaciones de la suite PARSEC sobre la jerarquía de memoria del procesador, utilizando las entradas pequeña, mediana, grande y nativa. La creencia generalizada sostiene que la entrada nativa es la más cercana a una ejecución real y presionará más la jerarquía de memoria, siendo el resto de entradas aproximaciones que resultarán menos precisas.

Los resultados obtenidos del análisis del instruction mix y el footprint de las entradas de cada aplicación apoyan la idea de que las entradas de menor tamaño son versiones reducidas de la entrada nativa que presionarán menos la jerarquía de memoria. Estos experimentos se han realizado ejecutando las aplicaciones con un único thread, aunque los resultados obtenidos para el instruction mix y el footprint serán válidos al aumentar el número de threads de la aplicación. La verificación de esta afirmación queda pendiente como trabajo futuro.

Observando el número de fallos de TLB nos damos cuenta de que no obtenemos necesariamente más fallos con las entradas de mayor tamaño, lo que nos hace empezar a pensar que es probable que las entradas nativas no ejerzan mayor presión sobre la jerarquía de memoria y crea la necesidad de realizar un estudio en mayor profundidad.

Se ha analizado la tasa de fallos en cache para todas las entradas de los programas variando

la capacidad de la cache tanto en una arquitectura Intel (utilizando VALGRIND) como en Sparc (mediante simulación con Simics). Además, se han obtenido trazas temporales de los fallos utilizando Simics. Por completitud de los resultados, también habría que añadir las estadísticas de las simulaciones que no han terminado a tiempo para la entrega del proyecto. A partir de estos resultados podemos concluir sin lugar a dudas que no se cumple la creencia popular que indica que las entradas mayores presentan más fallos en cache. Hay aplicaciones en las que el número de fallos disminuye a medida que aumenta el tamaño de la entrada, y otras en las que prácticamente no varía. En cualquier caso, no es cierto que la tasa de fallos sea notablemente más elevada para la entrada nativa que para el resto, lo cual prueba que aproximaciones como reducir significativamente el tamaño de la cache al utilizar una entrada de menor tamaño para intentar reproducir un comportamiento realista son incorrectas.

Como resultado final del proyecto, hemos presentado una selección de las entradas más adecuadas para cada aplicación para llevar a cabo un estudio de la jerarquía de memoria en un tiempo razonable, ya sea por ser más representativas de la entrada nativa o por presentar tasas de fallos más elevadas. En algunos casos se propone ejecutar únicamente una sección de la entrada nativa, en otros, se propone utilizar una de las entradas de menor tamaño y en otros, se indican los parámetros de una entrada nueva distinta de todas las que ya existen. Además, se señalas las aplicaciones que presentan un número de fallos excesivamente bajo resultando, por tanto, inadecuadas para estudiar la jerarquía de memoria.

El estudio de tasas de fallos y trazas temporales se ha realizado sobre un nivel de memoria cache ejecutando las aplicaciones con un solo thread, a pesar de que la suite PARSEC está destinada al análisis de multiprocesadores. Pensamos que nuestras conclusiones seguirán siendo válidas en un entorno multiprocesador, aunque influirá el tipo de paralelismo utilizado en cada aplicación. Esta hipótesis debería ser confirmada con un estudio que incluyera la variación en el número de threads y procesadores que queda planteado como trabajo futuro. De todas formas, usar nuestra selección de entradas ofrece mayores garantías de obtener resultados válidos que usar una entrada de menor tamaño simplemente por mantener un tiempo de simulación razonable.

6.2 Conclusiones a nivel personal

La experiencia del desarrollo de este proyecto me ha resultado muy positiva. Principalmente, me ha servido como introducción a la investigación. Me ha ayudado a decidir que me gustaría continuar con esta línea de trabajo y estudiar un máster y un doctorado dentro de la arquitectura de computadores.

He aprendido que en investigación es necesario tener muchas cosas en cuenta, y para eso resulta extremadamente útil compartir tus ideas con otras personas que pueden aportarte nuevos puntos de vista o indicarte qué se te ha olvidado considerar. Además, es importante tener claro en todo momento qué objetivos intentamos conseguir y planificar previamente lo que se va a hacer para no realizar trabajo inútil o redundante.

También he podido aplicar directamente los conocimientos adquiridos durante la carrera en las asignaturas de arquitectura de computadores. Además, las destrezas generales adquiridas a lo largo de los últimos años me han servido para aprender rápidamente las nuevas herramientas, adaptarme a la metodología de trabajo y resolver problemas con eficacia.

Bibliografía

[1] Gnuplot. http://www.gnuplot.info/ (Último acceso agosto 2011).

[2] Simulador gem5. http://www.gem5.org/Main_Page (Último acceso agosto 2011).

[3] Valgrind. http://valgrind.org/ (Último acceso agosto 2011).

[4] N. Agarwal, T. Krishna, Li-Shiuan Peh, and N.K. Jha. Garnet: A detailed on-chip network model inside a full-system simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 33 –42, 2009.

[5] Jesus Alastruey, Jose Luis Briz, Pablo Ibáñez, and Victor Viñals. Software demand, hardware supply. *IEEE Micro*, 26:72–82, 2006.

[6] T. Austin, E. Larson, and D. Ernst. Simplescalar: an infrastructure for computer system modeling. *Computer*, 35(2):59 –67, feb 2002.

[7] Nick Barrow-Williams, Christian Fensch, and Simon Moore. A communication characterization of splash-2 and parsec. In *Proceedings of the 2009 International Symposium on Workload Characterization*, October 2009.

[8] Major Bhadauria, Vincent M. Weaver, and Sally A. McKee. Understanding parsec performance on contemporary cmps. In *Proceedings of the 2009 International Symposium on Workload Characterization*, October 2009.

[9] Abhishek Bhattacharjee and Margaret Martonosi. Characterizing the tlb behavior of emerging parallel workloads on chip multiprocessors. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 29–40, Washington, DC, USA, 2009. IEEE Computer Society.

[10] C. Bienia and Kai Li. Fidelity and scaling of the parsec benchmark inputs. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1 –10, dec. 2010.

[11] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[12] Christian Bienia, Sanjeev Kumar, and Kai Li. Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *Proceedings of the 2008 International Symposium on Workload Characterization*, September 2008.

[13] 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*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[14] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26:52–60, July 2006.

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

[16] Blas A. Cuesta, Alberto Ros, María E. Gómez, Antonio Robles, and José F. Duato. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *Proceeding of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 93–104, New York, NY, USA, 2011. ACM.

[17] Universidad de Wisconsin. Condor. `http://www.cs.wisc.edu/condor/` (Último acceso agosto 2011).

[18] The Embedded Microprocessor Benchmark Consortium (EEMBC). Coremark. `http://www.coremark.org/home.php` (Último acceso agosto 2011).

[19] The Embedded Microprocessor Benchmark Consortium (EEMBC). Multibench. `http://www.eembc.org/benchmark/multi_sl.php` (Último acceso agosto 2011).

[20] M. Ekman and P. Stenstrom. Enhancing multiprocessor architecture simulation speed using matched-pair comparison. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, pages 89–99, Washington, DC, USA, 2005. IEEE Computer Society.

[21] Boris Grot, Joel Hestness, Stephen W. Keckler, and Onur Mutlu. Kilo-noc: a heterogeneous network-on-chip architecture for scalability and service guarantees. In *Proceeding of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 401–412, New York, NY, USA, 2011. ACM.

[22] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3 – 14, dec. 2001.

[23] Kyle C. Hale, Boris Grot, and Stephen W. Keckler. Segment gating for static energy reduction in networks-on-chip. In *Proceedings of the 2nd International Workshop on Network on Chip Architectures*, NoCArc '09, pages 57–62, New York, NY, USA, 2009. ACM.

[24] Intel. Contadores hardware para procesadores intel. `http://software.intel.com/sites/products/documentation/hpc/amplifierxe/en-us/lin/ug_docs/reference/index.htm` (Último acceso agosto 2011).

[25] Intel. Intel vtune amplifier xe. `http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/` (Último acceso agosto 2011).

[26] Intel. Pin. `http://www.pintool.org/` (Último acceso agosto 2011).

[27] Syed Muhammad Zeeshan Iqbal, Yuchen Liang, and Hakan Grahn. Parmibench - an open-source benchmark for embedded multiprocessor systems. *IEEE Comput. Archit. Lett.*, 9:45–48, July 2010.

[28] AJ KleinOsowski, John Flynn, Nancy Meares, and David J. Lilja. Adapting the spec 2000 benchmark suite for simulation-based computer architecture research. In Lizy Kurian John and Ann Marie Grizzaffi Maynard, editors, *Workload Characterization of Emerging Computer Applications*, volume 610 of *The Kluwer International Series in Engineering and Computer Science*, pages 83–100. Springer US, 2001. 10.1007/978-1-4615-1613-2_4.

[29] M. Lis, Pengju Ren, Myong Hyon Cho, Keun Sup Shim, C.W. Fletcher, O. Khan, and S. Devadas. Scalable, accurate multicore simulation in the 1000-core era. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 175 –185, april 2011.

[30] 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.

[31] 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, November 2005.

[32] J.E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1 –12, 2010.

[33] Asit K. Mishra, Aditya Yanamandra, Reetuparna Das, Soumya Eachempati, Ravi Iyer, N. Vijaykrishnan, and Chita R. Das. Raft: A router architecture with frequency tuning for on-chip networks. *J. Parallel Distrib. Comput.*, 71:625–640, May 2011.

[34] Angeles Navarro, Rafael Asenjo, Siham Tabik, and Călin Caşcaval. Load balancing using work-stealing for pipeline parallelism in emerging applications. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 517–518, New York, NY, USA, 2009. ACM.

[35] Sun Developer Network (SDN) ORACLE. Shade. `http://developers.sun.com/solaris/articles/shade.html` (Último acceso agosto 2011).

[36] SPEC. Spec omp, 2001. `http://www.spec.org/omp/` (Último acceso agosto 2011).

[37] SPEC. Spec cpu2006, 2006. `http://www.spec.org/cpu2006/` (Último acceso agosto 2011).

[38] Evangelos Vlachos, Michelle L. Goodstein, Michael A. Kozuch, Shimin Chen, Babak Falsafi, Phillip B. Gibbons, and Todd C. Mowry. Paralog: enabling and accelerating online parallel monitoring of multithreaded applications. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 271–284, New York, NY, USA, 2010. ACM.

[39] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM.

[40] David A. Wood, Mark D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. In *Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '91, pages 79–89, New York, NY, USA, 1991. ACM.

[41] R.E. Wunderlich, T.F. Wenisch, B. Falsafi, and J.C. Hoe. Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 84 – 95, june 2003.

Anexo A
Gestión del proyecto

Este anexo 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 marzo hasta agosto de 2011, en dedicación a tiempo completo. 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.

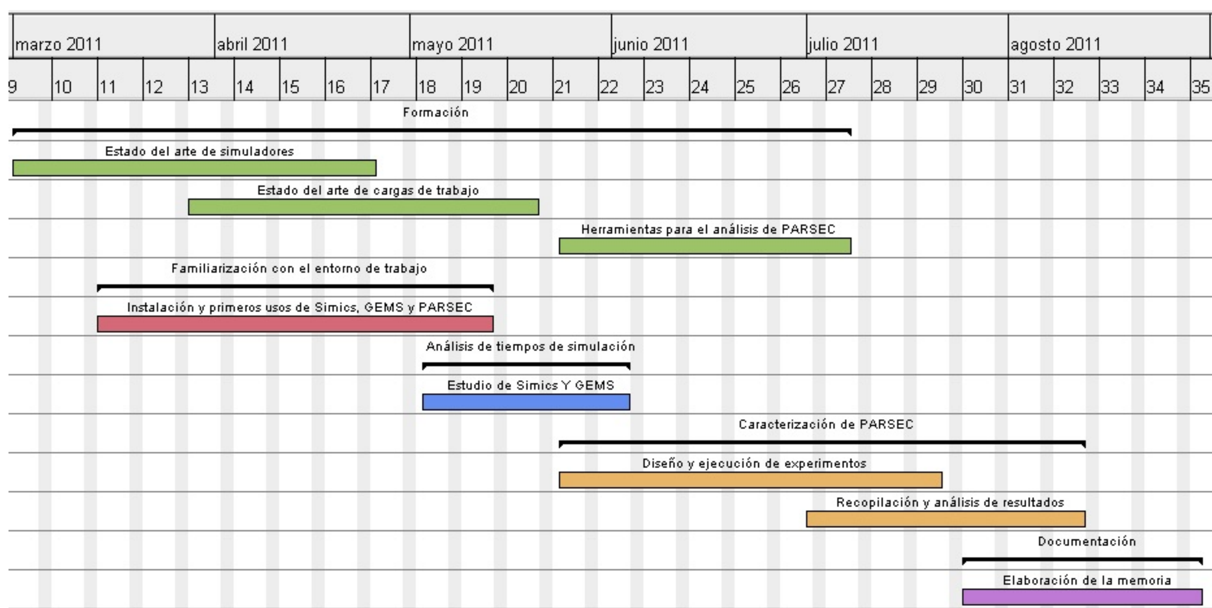


Figura A.1: Diagrama de Gantt del proyecto.

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

- **Formación.** Este proyecto tiene un importante componente de formación, ya que se han utilizado numerosas herramientas que no se conocían anteriormente. Por un lado, ha sido necesario estudiar el estado del arte tanto de los simuladores como de las cargas de trabajo. Por otro lado, para la caracterización de Parsec se ha aprendido a utilizar herramientas para la monitorización de programas y el estudio de prestaciones.

- **Familiarización con el entorno de trabajo**. Para comenzar a trabajar, instalamos y configuramos Simics y GEMS en nuestro ordenador local. Este proceso es bastante costoso ya que se trata de herramientas poco orientadas a los usuarios. También fue necesario instalar GEMS en nuestra carpeta dentro del cluster del departamento y hacer que funcionara con Simics, que estaba ya instalado. Además, se realizaron las primeras pruebas de simulación de los programas de PARSEC para comprender cómo funcionaba el simulador

- **Análisis de tiempos de simulación.** Durante esta parte del proyecto se realizó un estudio del tiempo de simulación utilizando Simics y GEMS y se consideró la posibilidad de optimizar del simulador.

- **Caracterización de PARSEC.** Esta es la tarea principal del proyecto. En ella se engloba el diseño y ejecución de los experimentos necesarios para estudiar el funcionamiento de las aplicaciones de PARSEC sobre la jerarquía de memoria y la recopilación y análisis de los resultados para obtener conclusiones.

- **Documentación.** Esta parte se corresponde con la redacción de la memoria en LaTeX. Se documentó también el proceso de instalación de Simics y GEMS para facilitar el trabajo a quienes deseen utilizarlo en un futuro.

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

En la realización del proyecto se han invertido un total de **704 horas**. En la figura A.2 se presenta la distribución de este tiempo en las diferentes tareas. Se ve claramente que la mayor parte del tiempo se ha invertido en la caracterización de PARSEC, que es la tarea principal, seguida de la formación, que se ha extendido a lo largo de prácticamente todo el proyecto.

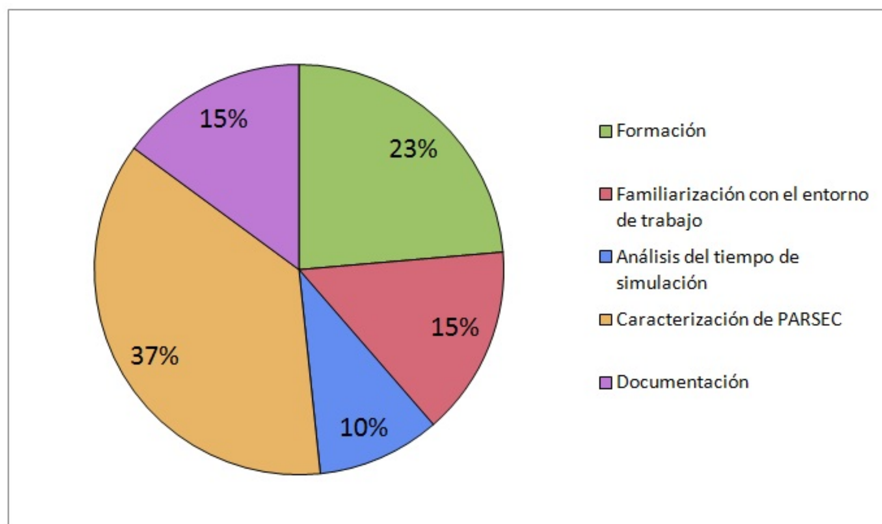


Figura A.2: Distribución del tiempo en las diferentes tareas del proyecto.

En la tabla A.1 se muestra de manera más detallada la cantidad de horas invertidas en las actividades que componen cada tarea.

Tarea	Número de horas
Formación	166.45
Estado del arte de simuladores	39.2
Estado del arte de cargas de trabajo	50.5
Herramientas para el análisis de PARSEC	76.75
Familiarización con el entorno de trabajo	105.5
Análisis de tiempos de simulación	68.5
Caracterización de PARSEC	258.8
Diseño y ejecución de experimentos	200.55
Recopilación y análisis de resultados	58.25
Documentación	105.05
NÚMERO TOTAL DE HORAS	**704.3**

Tabla A.1: Número de horas invertidas en cada una de las tareas del proyecto.

A.3 Problemas encontrados

Los principales problemas encontrados durante el desarrollo del proyecto surgieron en relación con las simulaciones y venían causados por su alto coste en tiempo y recursos. Un fallo que conllevara la cancelación de una o varias simulaciones, o su repetición una vez terminadas, podía suponer un retraso de varios días hasta que se disponía de los resultados correctos. Por otro lado, algunas de las simulaciones con entradas más grandes tardaban más de un mes en completarse, lo que ha imposibilitado tener todos los resultados para la fecha de entrega del proyecto. Además, a finales de julio y principios de agosto se apagó el cluster del departamento para realizar labores de mantenimiento. Como al lanzar muchas de las simulaciones no se sabía todavía que el cluster estaría inoperativo durante varios días, no se tuvo en cuenta este hecho y se tuvo que detener su ejecución sin posibilidad de guardar su estado para retomarlas más adelante.

Debido también a la cantidad de tiempo que tardaban algunas simulaciones no era posible lanzarlas de manera automática con Condor porque serían expulsadas tras varios días de ejecución, así que fue necesario distribuirlas de forma manual por los nodos del cluster. Como además necesitan mucha memoria RAM, podían lanzarse un número limitado de simulaciones al mismo tiempo, lo que alargaba más todavía la espera hasta disponer de todos los resultados.

A parte de eso, el gran tamaño de las entradas nativas hizo que fuera necesario crear una nueva arquitectura con mayor memoria para algunas de las aplicaciones, teniendo que repetir el proceso de inicio del sistema operativo, copia de los programas al sistema simulado y ejecución hasta el inicio de la región de interés. En uno de los casos se necesitaba un disco duro de mayor tamaño, lo cual se podía solucionar añadiendo otro disco a la configuración de partida. Como el tiempo que nos habría costado preparar el nuevo sistema era demasiado elevado, se decidió no realizar las simulaciones correspondientes a ese caso.

Anexo B
Análisis del tiempo de simulación: Simics y GEMS

En este anexo se detalla el estudio realizado del tiempo de simulación de multiprocesadores de memoria compartida con Simics y GEMS y la distribución de ese tiempo en los diferentes módulos del simulador. Se han utilizado aplicaciones pertenecientes a la suite PARSEC.

B.1 Tiempo de ejecución de las simulaciones

Para tener una idea más precisa de cuánto se ralentiza la ejecución de una aplicación al utilizarla en un simulador, comenzaremos midiendo los tiempos de simulación. Hemos utilizado Simics para realizar la simulación funcional y, posteriormente, hemos incorporado el módulo GEMS para llevar a cabo una simulación temporal, que tendrá en cuenta los detalles de la jerarquía de memoria. Estas herramientas han sido introducidas en la sección 2.1 y se explica el método utilizado para trabajar con ellas en la sección 4.5 y, más detalladamente, en el anexo C. Como aplicaciones se han utilizado las de la suite PARSEC (introducida en la sección 2.2 y explicada en mayor profundidad en el capítulo 3) que presentan más fallos en memoria según [8]: `canneal`, `fluidanimate` y `streamcluster`. Se han utilizado estas aplicaciones porque, al presentar más fallos en memoria, requerirán más trabajo por parte del simulador y de esta forma obtendremos una cota superior del tiempo de simulación para las aplicaciones de PARSEC. Se han llevado a cabo las simulaciones con las entradas pequeña, mediana y grande de cada una de las tres aplicaciones. Además, para comprobar cuánto aumenta el tiempo de ejecución al simular un número de procesadores mayor, se han realizado las pruebas para uno y dos procesadores. Cuando se utilizan dos procesadores, se ejecutan las aplicaciones con dos threads para aprovechar el paralelismo que presentan.

En la figura B.1 se presenta el tiempo de simulación para las tres entradas de las aplicaciones, con uno y dos procesadores, utilizando únicamente Simics y usando Simics junto con el módulo GEMS. Para facilitar la visualización de los datos se ha utilizado un eje logarítmico. Vemos que la simulación funcional tarda aproximadamente entre 10 segundos y 15 minutos, mientras que la simulación temporal tarda entre 30 minutos y 42 horas. El tiempo de simulación es similar para uno y dos procesadores, siendo en general un poco superior para dos procesadores. Esto se debe a que, como se trata de programas muy escalables, Simics simula el mismo código aunque lo reparta en varios procesadores. La simulación con más procesadores tarda más tiempo en la mayoría de los casos debido a la sobrecarga de sincronización de la aplicación y al aumento de la complejidad de los elementos que debe gestionar el simulador. La única excepción es la aplicación `canneal`, en la que la simulación de dos procesadores tarda menos que la de un procesador. Al duplicar el número de threads, este programa tarda en ejecutarse

menos de la mitad del tiempo (de 32 a 14 segundos para la entrada grande, por ejemplo), por lo tanto el tiempo total simulado al utilizar dos threads sumando el de los dos procesadores es menor que el tiempo de simular con un thread, así que es lógico que la simulación tarde menos tiempo.

También podemos comprobar, como ya imaginábamos, que el tiempo de simulación aumenta con el tamaño de la entrada. Al ir pasando de pequeña a mediana y de mediana a grande el tiempo se multiplica por un factor que tiene un valor entre 2 y 5 en las distintas aplicaciones.

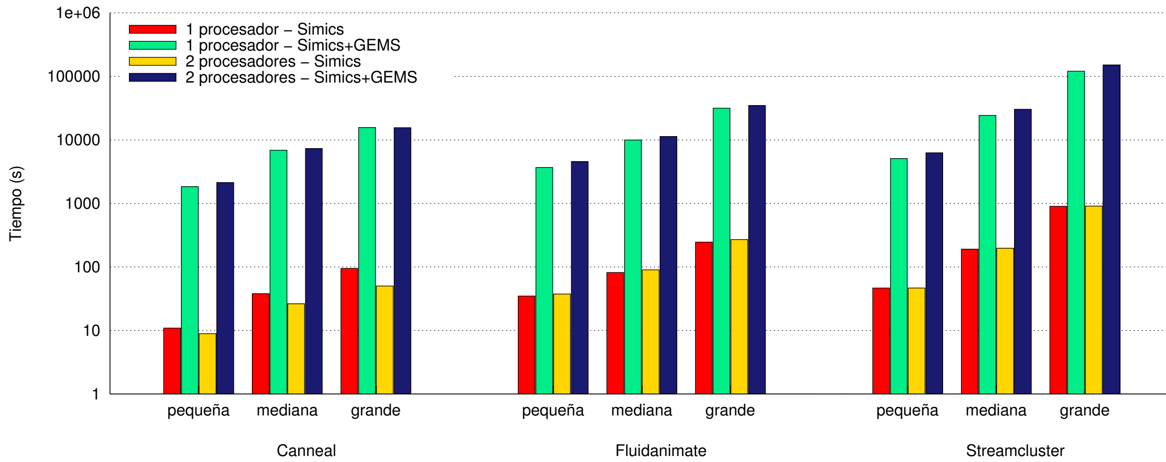


Figura B.1: Tiempo de simulación de `canneal`, `fluidanimate` y `streamcluster` con Simics y Simics+GEMS, con 1 y 2 procesadores.

El *slowdown* es la medida de cuántas veces más lenta resulta la simulación de la aplicación respecto de su ejecución nativa y puede calcularse como el tiempo que tarda la simulación partido por el tiempo simulado. En la figura B.2 vemos que, para la simulación funcional con un procesador, el slowdown se encuentra entre 1.5 y 3. Esto significa que simular la aplicación es entre 1.5 y 3 veces más lento que ejecutarla en nativo. Cuando utilizamos también el módulo GEMS para realizar simulación funcional el slowdown pasa a valer entre 100 y 135. Si pasamos a dos procesadores se ve claramente un aumento de estos valores, que están entre 2.5 y 5 para simulación funcional y entre 240 y 400 para la temporal, multiplicándose incluso por más de dos en varios casos. Cuando tenemos una arquitectura con varios procesadores, por cada ciclo del sistema simulado Simics debe simular varios, uno por cada procesador. Como la simulación siempre se ejecuta en serie, cuanto más aumente el número de procesadores, más aumentará el slowdown, llegando a ser impracticable la simulación de cien o más procesadores.

B.2 Distribución del tiempo en los diferentes módulos durante la ejecución de la simulación

Como el tiempo que tardan Simics y GEMS en simular las aplicaciones es tan grande que no permite la exploración eficiente de múltiples diseños para nuevas arquitecturas, decidimos realizar un estudio más detallado del comportamiento del simulador para intentar detectar cuellos de botella que pudieran ser optimizados. En esta sección presentamos las simulaciones realizadas con ese objetivo y los resultados obtenidos.

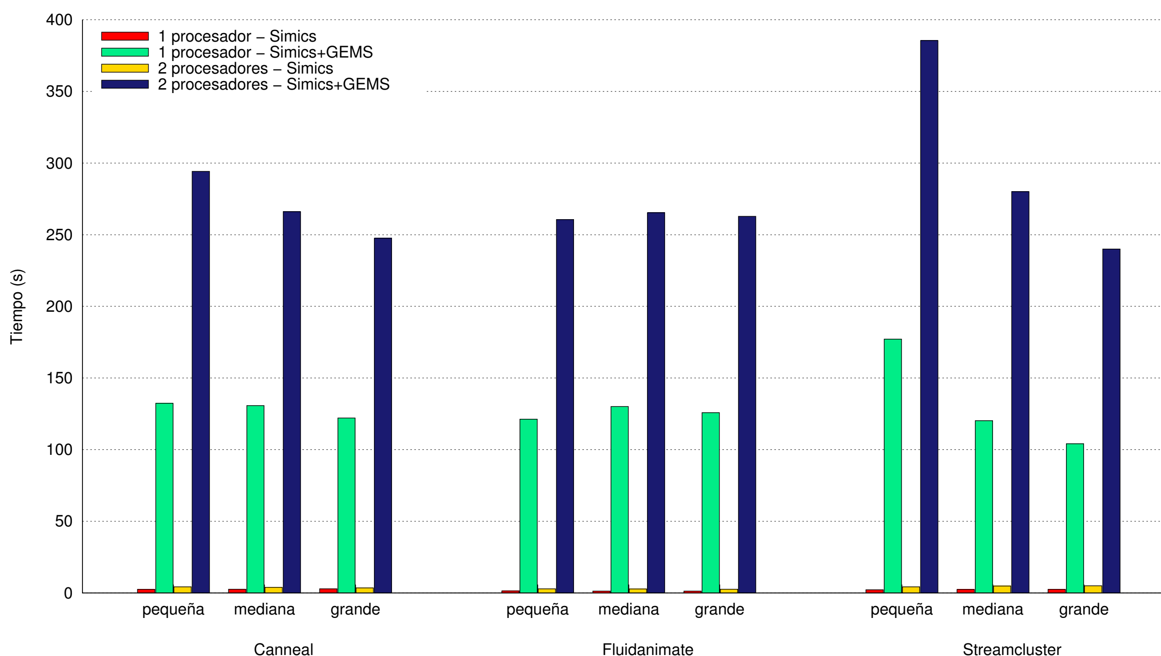


Figura B.2: Slowdown de las simulaciones de `canneal`, `fluidanimate` y `streamcluster` con Simics (izquierda) y Simics+GEMS (derecha), con 1 y 2 procesadores.

B.2.1 Tipos de simulaciones realizadas

Para esta parte del estudio decidimos utilizar los benchmarks `blackscholes`, `bodytrack` y `canneal` de PARSEC. `Blackscholes` es la aplicación que menos fallos presenta en cache, logra el paralelismo dividiendo la cantidad de trabajo entre los threads disponibles y tiene poca compartición de datos. `Bodytrack` tiene un *working set* mayor y el trabajo se distribuye por lo procesadores mediante un *pool* de threads. `Canneal`, también utilizado en la sección anterior, es la aplicación que más fallos presenta en cache, tiene un working set que podemos considerar ilimitado y una paralelización de grano fino poco estructurada. En todos los casos se ha utilizado la entrada pequeña de los benchmarks para mantener un tiempo de ejecución razonable. Se han realizado simulaciones de un arquitectura UltraSPARC III Plus con dos, cuatro y ocho procesadores, y sistema operativo Solaris 10.

Para poder obtener las conclusiones necesarias se han llevado a cabo los siguientes tipos de simulaciones:

- **Simics**: Simulación funcional utilizando únicamente Simics.
- **Simics+GEMS Ideal, latencia 0**: Simulación utilizando Simics y GEMS, pero usando GEMS para simular una jerarquía ideal. Esto significa que GEMS no hace realmente ningún cálculo, simplemente devuelve latencia cero para cualquier acceso a memoria. De esta manera estamos introduciendo únicamente el retraso correspondiente a la interacción entre Simics y GEMS.
- **Simics+GEMS Ideal, latencia realista**: Simulación utilizando Simics y GEMS, pero en este caso GEMS devolverá una latencia más realista, obtenida de la latencia media de otra simulación. Ha sido necesario indicar al simulador que debería devolver latencias decimales para que los valores se ajustaran a nuestras necesidades. GEMS sólo soportaba

el uso de números enteros como latencias, así que ha sido necesario añadir nuevas opciones de configuración y modificar el código para soportar la nueva funcionalidad. Utilizando este método logramos un CPI igual al de la simulación que estábamos intentando imitar.

- **Simics+Gems, L1 grande**: Simulación utilizando Simics y GEMS, con una cache de primer nivel grande, pretendiendo que la aplicación pueda disponer de todos los datos en la L1 y no tengan que hacerse reemplazos y accesos a la L2. En concreto, se ha utilizado una L1 separada para instrucciones y datos, cada una con 512KB y asociatividad 16. Habrá una L2 en cada procesador, con asociatividad 16, sumando en todos los casos 2MB.
- **Simics+Gems, L1 pequeña**: Simulación utilizando Simics y GEMS con la intención de ver cómo las aplicaciones se comportarían en una ejecución real, con un número de fallos apreciable en la L1. Como las entradas utilizadas son de pequeño tamaño, no es posible conseguir ese efecto con una L1 de tamaño normal, así que se ha realizado la simulación con una L1 excesivamente pequeña. Esta aproximación ha sido utilizada por Cuesta *et al.* en [16]. El tamaño utilizado para la L1 es de 1KB, con asociatividad 2. Para la L2 se usa la misma configuración que en caso anterior. Como se ha considerado que este era el caso más representativo, se han utilizado las latencias medias de acceso a memoria de estas simulaciones para las pruebas de Simics+Gems ideal con latencia realista. En los estudios realizados posteriormente, hemos podido comprobar que esta aproximación no es adecuada para reproducir el comportamiento de una ejecución realista. De todas formas, nos sirve para analizar el comportamiento del simulador ante una situación extrema en la que habrá un número muy elevado de fallos en cache.

Se han realizado las ejecuciones completas con Simics y GEMS para obtener los tiempos totales de simulación. Después, se ha utilizado VTune (explicado en profundidad en la sección 4.3) para obtener el porcentaje del tiempo de ejecución que pertenece a cada módulo. Para este análisis, VTune utiliza la técnica llamada muestreo basado en tiempo (time based sampling o TBS), que consiste en interrumpir la ejecución cada cierto tiempo y anotar en qué instrucción se encuentra el programa, informándonos al final de las zonas de código en las que la ejecución ha pasado más tiempo. Para esta parte se han ejecutado sólo 600 segundos de la simulación, habiendo comprobado que el resultado obtenido era el mismo que teniendo en cuenta le ejecución completa.

Los tiempos de simulación obtenidos siguen confirmando las conclusiones que de la sección B.1, dejando claro que se cumplen también con un mayor número de procesadores. El slowdown continúa aumentando linealmente con el número de procesadores, llegando a tomar valores superiores a 1000 cuando simulamos un sistema con ocho procesadores.

B.2.2 Resultados de la distribución de tiempos

Mostramos en las figuras B.3, B.4 y B.5 los tiempos de simulación con su distribución en los diferentes módulos para las simulaciones de todos los tipos con dos, cuatro y ocho procesadores. No aparecen en las gráficas las simulaciones en las que se utiliza Simics únicamente debido a que el tiempo de ejecución es mucho menor que el del resto de simulaciones. El caso en que se utiliza una cache L1 de tamaño muy pequeño se ha representado en la gráfica de la derecha, con un rango mayor para el eje y para que toda la información se visualizara mejor. Los experimentos de los benchmarks `bodytrack` y `canneal` con ocho procesadores y cache L1 pequeña no se han podido realizar porque se obtenían errores durante la simulación. Pensamos que esto se debe a que la cache es demasiado pequeña y el simulador no puede gestionar los accesos correctamente.

No se han realizado tampoco las simulaciones con memoria ideal que devuelven una latencia de acceso a memoria realista, ya que esta latencia ha sido obtenida en todos los casos a partir de la simulación con la cache L1 pequeña.

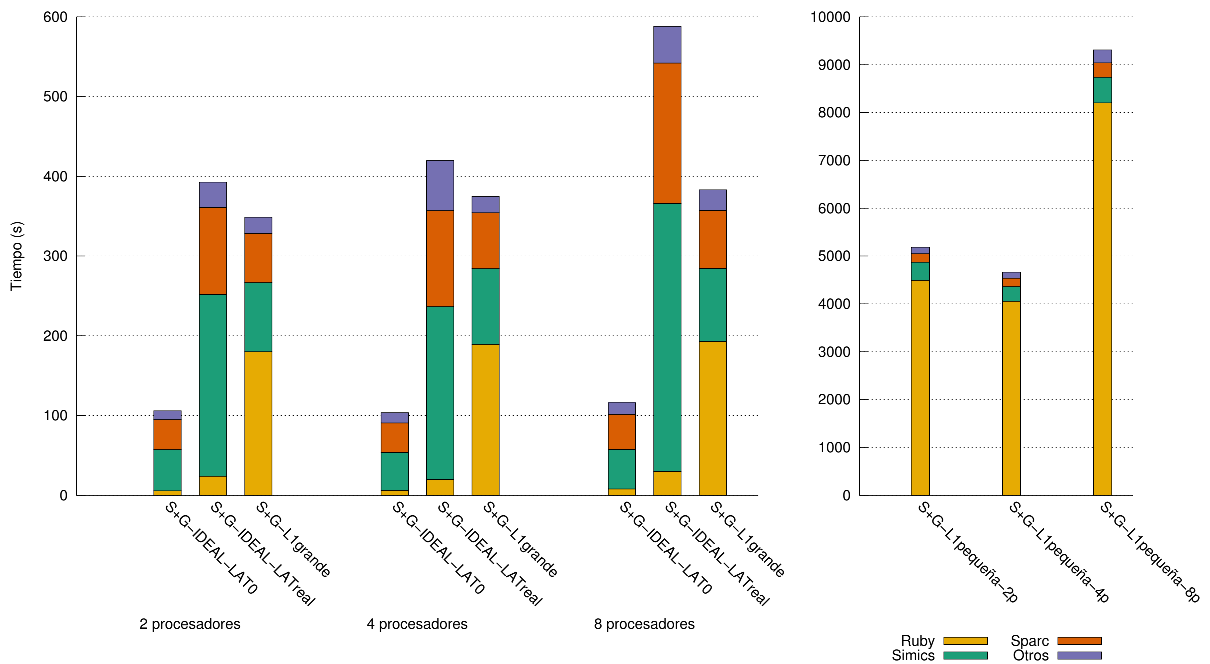


Figura B.3: Distribución del tiempo en los diferentes módulos simulando `blackscholes` con Simics y GEMS, con 2, 4 y 8 procesadores.

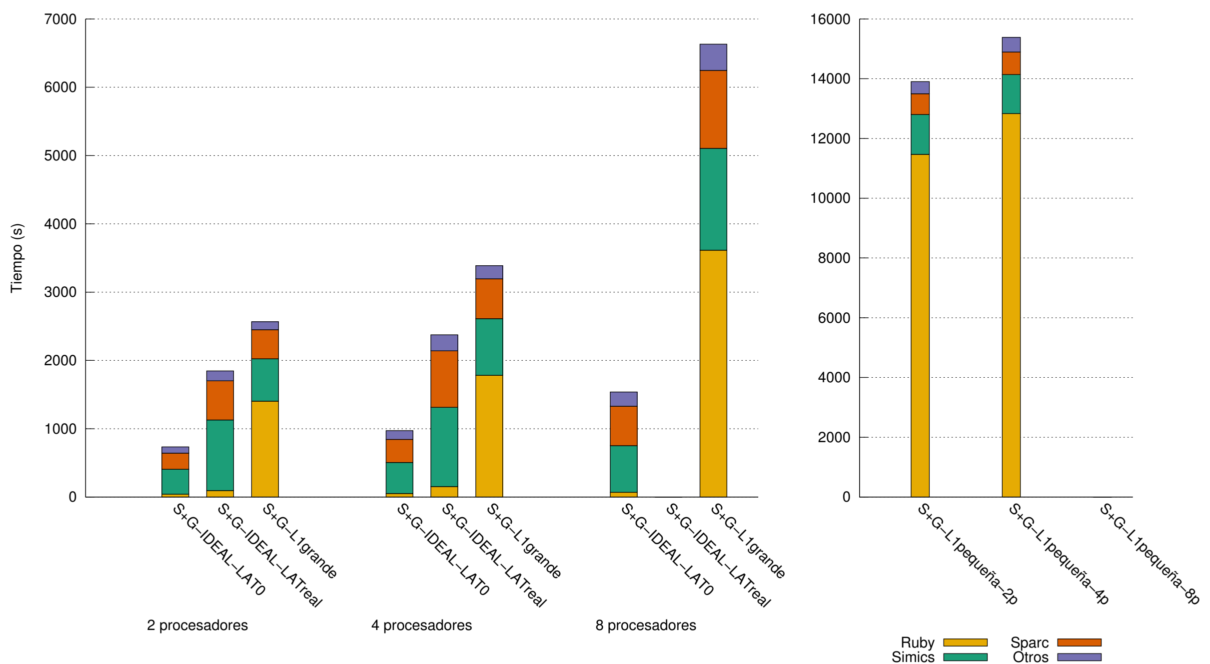


Figura B.4: Distribución del tiempo en los diferentes módulos simulando `bodytrack` con Simics y GEMS, con 2, 4 y 8 procesadores.

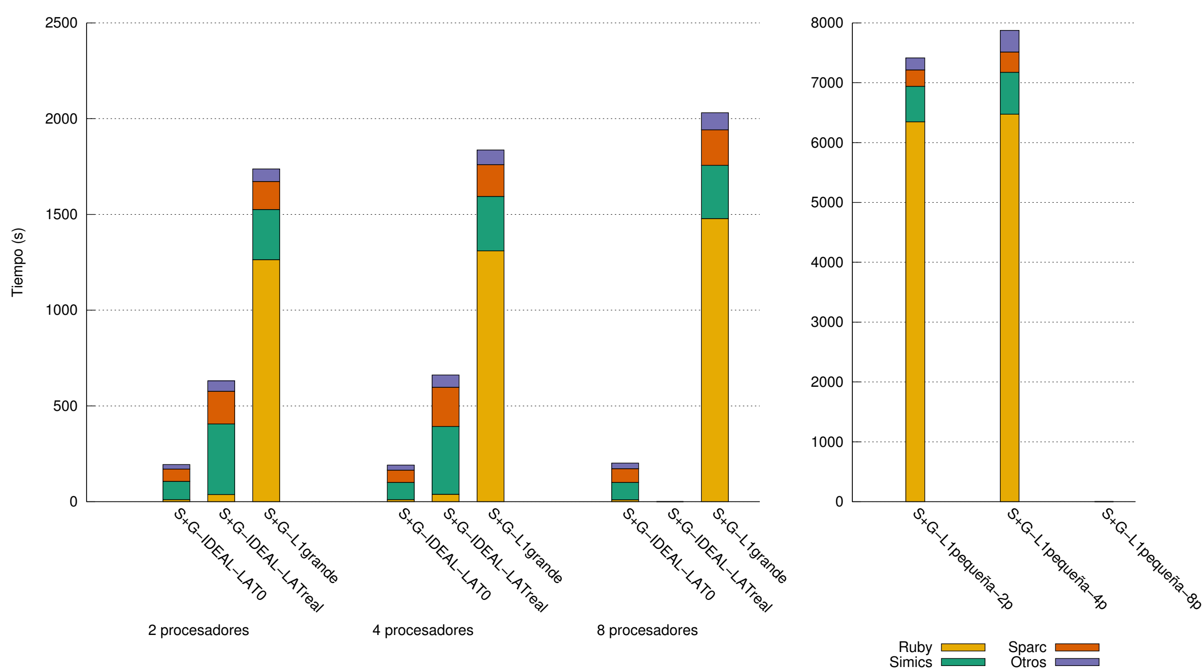


Figura B.5: Distribución del tiempo en los diferentes módulos simulando `canneal` con Simics y GEMS, con 2, 4 y 8 procesadores.

Centrándonos únicamente en el tiempo total de simulación podemos comenzar diciendo que el tiempo utilizando únicamente Simics es mucho menor que si utilizamos también GEMS, aunque este módulo no haga ningún trabajo y devuelva siempre latencia cero, lo que básicamente es lo mismo que no simular la jerarquía de memoria. Al utilizar GEMS, Simics le pasa la información de cada acceso a memoria y lo despierta cada ciclo para que realice los cálculos necesarios. Además, cuando la latencia devuelta por GEMS no es cero, el programa tardará más ciclos en ejecutarse provocando que la simulación tarde más tiempo.

Pensamos que el tiempo de ejecución en estos casos no aumenta por la comunicación que se realiza con GEMS cada ciclo, ya que se han hecho pruebas disminuyendo la frecuencia de esta comunicación y no han mejorado los resultados. Tampoco viene causado por el trabajo de GEMS, porque el tiempo de ejecución aumenta bastante aunque usemos memoria ideal. Por lo tanto, hemos concluido que la razón son los retrasos que introduce Simics ejecutando más ciclos, aunque esté esperando el resultado de un acceso a memoria. Desafortunadamente, no podemos profundizar más en este punto porque el código de Simics no es libre.

Por otro lado, al simular la jerarquía de memoria y coherencia utilizando GEMS, el tiempo de simulación es también mayor, principalmente al utilizar una L1 pequeña. Esto se debe claramente al aumento de trabajo que tiene que realizar GEMS. La única excepción es la aplicación `blackscholes`, en la que tarda más la simulación cuando utilizamos una memoria idea devolviendo latencia realista que cuando simulamos toda la jerarquía de memoria con una cache L1 grande. Cuando devolvemos una latencia realista la ejecución de la aplicación tarda un mayor número de ciclos porque se introducen las esperas por los datos de memoria, haciendo que aumente también el tiempo de simulación. Por lo tanto, si la latencia devuelta es muy grande el tiempo de simulación puede llegar a ser mayor que en el caso en que GEMS tiene que realizar trabajo pero devuelve latencias menores.

Si nos fijamos ahora en la distribución del tiempo vemos que aparecen claramente tres módulos principales: Simics, que se ocupa de la simulación funcional, Ruby (el módulo de GEMS que se ocupa de la jerarquía de memoria), para la simulación temporal, y Sparc, que es la arquitectura que estamos simulando. En un primer vistazo, nos damos cuenta de que la distribución general no cambia cuando variamos el benchmark o el número de procesadores.

Cuando utilizamos GEMS con memoria ideal vemos que el porcentaje de tiempo que se debe a Ruby es muy pequeño, aunque la parte de Simics y Sparc aumenta si la latencia devuelta es mayor, ya que el número de ciclos de ejecución aumenta. El peso de Ruby en la ejecución es pequeño porque no está haciendo trabajo real, únicamente devuelve la latencia que le hemos indicado. Cuando usamos GEMS para simular la jerarquía de manera realista vemos que comienza a ser una parte muy importante del total de la ejecución. En este punto, detectamos también que el porcentaje del tiempo de ejecución correspondiente a Ruby es mayor para `canneal` que para el resto de aplicaciones. Esto se debe a que `canneal` es la aplicación más fallos en cache presenta. Utilizando una cache L1 extremadamente pequeña vemos que Ruby supone más del 80 % del tiempo total de simulación. De todas formas, tras los estudios que se presentan en profundidad en el anexo D sabemos que esta ejecución no se aproxima a una ejecución real, porque los fallos en una cache extremadamente pequeña con la entrada pequeña de las aplicaciones no son equivalentes a los de la entrada nativa con una cache de tamaño común.

Como el módulo Ruby supone una gran parte del tiempo de ejecución y el código es libre, se ha realizado un análisis más detallado para intentar localizar si hay una función o parte del código en la que se invierta mucho tiempo. En la tabla B.1 se muestran los porcentajes de ejecución de los ficheros y funciones con mayor peso para la simulación de `blackscholes` utilizando Simics y GEMS con la cache L1 pequeña y 4 procesadores. De todas formas, como ya hemos comentado antes, los valores no variarán de manera relevante si tomamos otra aplicación o cambiamos el número de procesadores.

Fichero	**Porcentaje**	**Función**	**Porcentaje**
PerfectSwitch	15.5 %	PerfectSwitch::wakeup	15.5 %
Set	14.8 %	Set:setSize	5.4 %
		Set:count	5.0 %
		Set:operator=	1.9 %
		otros	2.4 %
Throttle	5.3 %	Throttle:wakeup	5.3 %
Vector	15.5 %	Vector<Set>:operator=	6.9 %
		Vector<Set>:grow	5.0 %
		Vector<Set>:Vector	2.2 %
		otros	3.4 %

Tabla B.1: Distribución del tiempo de ejecución en ficheros y funciones dentro del módulo Ruby durante una simulación del benchmark `blackscholes` con Simics y GEMS, con 4 procesadores.

Se puede ver que consumen mucho tiempo los módulos Set y Vector, pero son los proporcionados por C y su alto porcentaje se debe a que se utilizan en varios lugares diferentes y

las funciones son llamadas muchas veces. La función en la que más tiempo pasa la simulación, `PerfectSwitch::wakeup`, supone únicamente un 15.5 % del tiempo total de Ruby, es decir, aproximadamente un 13 % del total de la simulación. Por lo tanto, aunque se lograra que fuera más eficiente, la mejora no se reflejaría en el resultado global de manera importante. La clase `Throttle` se utiliza para controlar el ancho de banda a la salida de un router.

A pesar de todo, se ha hecho un pequeño análisis del grafo de llamadas para la función `PerfectSwitch::wakeup`, en la que deberían centrarse nuestros esfuerzos si deseáramos optimizar el simulador. En la tabla B.2 se presenta una lista de las funciones en el orden en que van llamándose unas a otras (la función 1 llama la función 2, la 2 a la 3, y así sucesivamente), la clase a la que se pertenecen y una pequeña descripción de las mismas.

Grafo de llamadas para la función `PerfectSwitch::wakeup`			
Orden	**Función**	**Clase o fichero**	**Descripción**
1	`runRubyEventQueue`	`interface.c`	Se ejecuta cada vez que Simics despierta a Ruby, en nuestro caso, cada ciclo.
2	`triggerEvents`	`EventQueue`	Se encarga de la pila de eventos pendientes.
3	`triggerWakeup`	`PerfectSwitch`	`PerfectSwitch` utiliza la clase virtual `Consummer`, que se ocupa de objetos que pueden ser objetivo de *wakeup calls*. La función se ocupa simplemente de llamar a la función `wakeup()`.
4	`wakeup`	`PerfectSwitch`	`PerfectSwitch` es un switch perfecto que no tiene latencia y utiliza una tabla de routing. Lo que más tiempo consume es el uso de *round robin* para elegir entre los puertos de entrada, mirar qué mensajes están esperando y utilizar *adaptive routing*.

Tabla B.2: Grafo de llamadas para la función en la pasa más tiempo una simulación con Simics y GEMS, `PerfectSwitch::wakeup`.

Vemos que se trata de una función que se llama varias veces durante todos los ciclos simulados. Para hacerla más eficiente sería necesario revisar tanto las estructuras de datos utilizadas como los algoritmos, y la mejora que podría lograrse tampoco sería muy notable respecto del total del tiempo de simulación.

B.3 Conclusiones

Podemos concluir que las simulaciones resultan muy lentas, aumentando el slowdown con el número de procesadores del sistema que simulamos. A pesar de que el tiempo de ejecución del módulo Ruby de GEMS supone una gran parte de la ejecución, dentro del módulo la distribución de tiempos es muy dispersa. No hay por tanto una parte del simulador que constituya un cuello de botella, sino que se utilizan muchas funciones que contribuyen con porcentajes pequeños al tiempo total de la simulación, dificultando en gran medida la optimización.

Anexo C
Detalles de las simulaciones con Simics y GEMS

En este anexo se explican en más profundidad las fases seguidas para llevar a cabo los experimentos, en especial las simulaciones con Simics y GEMS. Se incluyen también detalles del proceso de simulación y recogida de resultados.

C.1 Fases de desarrollo de los experimentos.

Para todos los experimentos realizados se han seguido las fases que se detallan en la figura C.1.

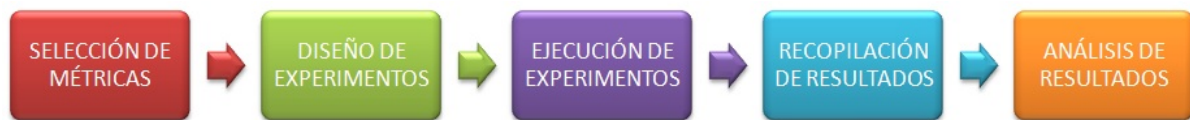


Figura C.1: Fases de realización de experimentos

A continuación se incluye una pequeña explicación de cada fase:

1. **Selección de métricas.** Antes de comenzar hay que decidir qué métricas necesitaremos obtener para realizar el estudio que nos interesa y obtener conclusiones. Las métricas que hemos utilizado aparecen detalladas en la sección 4.1.

2. **Diseño de experimentos.** En este punto hay que elegir qué herramientas se utilizarán para calcular las métricas ya seleccionadas y cómo se utilizará cada una de ellas. En el capítulo 4 se explican todas las herramientas y para qué se ha utilizado cada una.

3. **Ejecución de los experimentos.** Si, como en nuestro caso, el número de experimentos es muy grande, habrá que automatizar el proceso de ejecución para que resulte más cómodo y se aproveche el tiempo de CPU al máximo.

4. **Recopilación de resultados.** Una vez que se han ejecutado los experimentos, es necesario recoger la información que hemos obtenido, seleccionar lo que nos interesa y representarlo de forma adecuada.

5. **Análisis de resultados.** Por último, hay que analizar los resultados para obtener conclusiones. Es posible que en este punto detectemos nuevas necesidades y tengamos que repetir

el proceso de nuevo. Los resultados de nuestro trabajo se presentan en el capítulo 5 y, con mayor detalle, en el anexo D.

En el resto del anexo nos centraremos en el diseño y ejecución de las simulaciones y la recopilación de los resultados, ya que es la parte más compleja del trabajo realizado y la que más interesa comentar en profundidad. Las simulaciones realizadas a las que se hará referencia ya han sido explicadas en la sección 4.5 y en el anexo B.

C.2 Diseño de las simulaciones

Para realizar las simulaciones con Simics, el primer paso es definir los parámetros de la arquitectura e iniciar el sistema operativo. Después, ejecutamos las aplicaciones usando simulación funcional y creamos un checkpoint justo antes de comenzar la región de interés. Para todo lo que vamos a explicar a continuación supondremos que partimos de ese checkpoint.

Para trabajar con Simics, lo más cómodo es crear un script de ejecución y pasárselo al programa para que todas las acciones se lleven a cabo de manera automatizada. Lo más sencillo y rápido es realizar una ejecución funcional utilizando Simics. A continuación se presenta un script que se ocupa de la simulación de la región de interés del benchmark `blackscholes` en cuatro procesadores:

```
read-configuration "proc004-parsec-blackscholes-small-ready.check"
con0.capture-start "output.txt"
magic-break-enable
continue
continue
con0.capture-stop
quit
```

El comando `read-configuration` se utiliza para cargar el checkpoint que habíamos creado anteriormente. Si deseamos almacenar la salida que aparece en la consola del sistema simulado podemos utilizar `con0.capture-start` para comenzar a escribirla en un fichero y `con0.capture-stop` para terminar. `magic-break-enable` se utiliza para habilitar las instrucciones especiales llamadas *magic-instructions* que pararán la ejecución automáticamente. PARSEC viene preparado con estas instrucciones para que, al simular las aplicaciones en Simics, la simulación se detenga justo antes y después de la región de interés. El comando `continue` inicia la simulación, y debemos ejecutarlo dos veces porque la primera vez parará debido a la magic-instruction que marca el inicio de la ROI. La siguiente vez que se detenga la simulación ya habrá finalizado la región de interés y sólo tendremos que parar de escribir en el fichero de salida y finalizar la ejecución con `quit`.

Para ejecutar la simulación es suficiente con escribir en la línea de comandos

```
./simics -stall -no-win miScript.simics
```

siendo `miScript.simics` el script que presentábamos antes. La opción `-stall` indica el modo de simulación, que en este caso permite que el sistema pare la ejecución y es obligatorio cuando deseamos utilizar GEMS. La opción `-no-win` puede incluirse si no queremos que se abra una nueva ventana con la consola del sistema simulado.

Para incorporar la simulación temporal y el funcionamiento detallado de las caches podemos utilizar el módulo Ruby de GEMS, tal y como hemos hecho en el estudio de tiempos de simulación explicado en el anexo B. Antes de comenzar la región de interés habrá que indicar los parámetros de configuración para el módulo y, al terminar, se pueden volcar las estadísticas en un fichero para consultarlas posteriormente. Presentamos a continuación el script utilizado para una ejecución con Ruby:

```
read-configuration "proc004-parsec-blackscholes-small-ready.check"
con0.capture-start "output.txt"
magic-break-enable

# Iniciamos gems
add-module-directory ../../amd64-linux/lib
instruction-fetch-mode instruction-fetch-trace
istc-disable
dstc-disable
cpu-switch-time 1
load-module ruby
ruby0.setparam g_NUM_PROCESSORS 4
ruby0.setparam g_PROCS_PER_CHIP 4
ruby0.setparam g_MEMORY_SIZE_BYTES 4294967296
ruby0.setparam g_NUM_L2_BANKS 2
ruby0.setparam g_NUM_MEMORIES 1
ruby0.setparam L1_CACHE_ASSOC 16
ruby0.setparam L1_CACHE_NUM_SETS_BITS 9
ruby0.setparam L2_CACHE_ASSOC 16
ruby0.setparam L2_CACHE_NUM_SETS_BITS 10
ruby0.setparam SIMICS_RUBY_MULTIPLIER 1
ruby0.setparam_str PROTOCOL_DEBUG_TRACE false
ruby0.setparam L1CACHE_TRANSITIONS_PER_RUBY_CYCLE 1
ruby0.setparam L2CACHE_TRANSITIONS_PER_RUBY_CYCLE 1
ruby0.setparam DIRECTORY_TRANSITIONS_PER_RUBY_CYCLE 1
ruby0.setparam_str FINITE_BUFFERING true

ruby0.init

continue
continue
ruby0.dump-stats proc004-parsec-blackscholes-small.stat
con0.capture-stop
quit
```

El comando `add-module-directory` sirve para indicarle a simics dónde está el módulo que deseamos usar, en caso de que no lo encuentre. Con el comando `instruction-fetch-mode instruction-fetch-trace` Simics dirigirá al sistema de memoria los accesos correspondientes a búsqueda de instrucciones, que por defecto serían ignorados. Para acelerar el proceso de simulación, Simics utiliza *Simulator Translation Caches* (STCs) para evitar realizar todos los accesos a través del espacio de memoria. Como en este caso nos interesará utilizar Ruby para los accesos a memoria, desactivamos las STCs mediante los comandos `istc-disable` y

`dstc-disable`. Con el comando `cpu-switch-time 1` indicamos que nos interesa que Simics simule un ciclo de cada uno de los cuatro procesadores por turno. A continuación cargamos el módulo Ruby y lo configuramos, siendo la mayor parte de los parámetros autoexplicativos. Con la opción `SIMICS_RUBY_MULTIPLIER` indicamos cada cuántos ciclos queremos que Simics se comunique con Ruby. Después de configurar el módulo, iniciamos Ruby mediante el comando `ruby0.init`, y, tras la región de interés, obtenemos las estadísticas con `ruby0.dump-stats`.

En las simulaciones presentadas en la sección 4.5 no se utiliza GEMS para simular la jerarquía de memoria, sino que se usa una cache proporcionada por Simics. El caso más complejo es el de las simulaciones para realizar la traza temporal de las entradas nativas, en las que se tomaban diez muestras de la ejecución durante las cuales se obtenían las estadísticas cada 10 millones de ciclos. La primera parte del script utilizado para la simulación de ese tipo del benchmark `blackscholes` es la siguiente:

```
magic-break-enable
read-configuration "proc001-parsec-blackscholes-native-ready.check"
continue

# Iniciamos la cache
@cache = pre_conf_object("cache","g-cache")
@cache.cpus = conf.cpu0
@cache.config_line_number = 1024
@cache.config_line_size = 64
@cache.config_assoc = 8
@cache.config_virtual_index = 0
@cache.config_virtual_tag = 0
@cache.config_replacement_policy = "lru"
@cache.penalty_read = 0
@cache.penalty_write = 0
@cache.penalty_read_next = 0
@cache.penalty_write_next = 0
@cache.config_write_allocate = 1
@cache.config_write_back = 1
@SIM_add_configuration([cache],None)
phys_mem->timing_model = cache

@def hap_Mode_Switch(data, object, old_mode, new_mode):
        if (new_mode == Sim_CPU_Mode_Supervisor):
                # Remove Model
                run_command("phys_mem->timing_model = 0")
        elif (new_mode == Sim_CPU_Mode_User):
                # Attach Model
                run_command("phys_mem->timing_model = cache")

# Register callback
@SIM_hap_add_callback("Core_Mode_Change", hap_Mode_Switch, None)
```

```
# MUESTRA NUMERO 1

echo "\n\n MUESTRA NUMERO 1\n\n"

$iteration = 0
while $iteration < (2950/10) {
cache.reset-statistics
run-cycles 10000000
echo "\n\n iteration "
echo $iteration
ptime
cache.statistics
print-statistics -all
$iteration += 1
}
write-configuration finMuestra01.check

# MUESTRA NUMERO 2

# Desactivo cache y función callback
phys_mem->timing_model = 0
@SIM_hap_delete_callback("Core_Mode_Change", hap_Mode_Switch, None)

#Avanzo los ciclos que quiero saltar
run-cycles 1040000000

write-configuration inicioMuestra02.check

#Caliento caches
phys_mem->timing_model = cache
run-cycles 100000000

@SIM_hap_add_callback("Core_Mode_Change", hap_Mode_Switch, None)

echo "\n\n MUESTRA NUMERO 2\n\n"

$iteration = 0
while $iteration < (2950/10) {
cache.reset-statistics
run-cycles 10000000
echo "\n\n iteration "
echo $iteration
ptime
cache.statistics
print-statistics -all
$iteration += 1
}
write-configuration finMuestra02.check
...
```

Para iniciar la cache comenzamos dando valores a los diferentes parámetros, que son autoexplicativos, y después asignamos la cache al modelo temporal para que Simics la tenga en cuenta al ir ejecutando las instrucciones. A continuación, definimos una función en Python que activará y desactivará la cache cuando pasemos de ejecutar código de sistema (modo protegido o supervisor) a código de usuario (modo usuario) y viceversa. De esta forma, únicamente las instrucciones correspondientes a código de usuario pasarán por la cache.

Después, empezamos a tomar las muestras. La primera la tomamos siempre al inicio de la región de interés. Como deseamos que cada muestra tenga la longitud de la entrada grande, que en este caso era 2950 millones de instrucciones, utilizaremos un bucle `while`. Dentro de cada iteración del bucle, ponemos a cero los contadores de estadísticas de la cache mediante el comando `cache.reset-statistics`, ejecutamos los siguientes 10 millones de ciclos con `run-cycles` y mostramos las estadísticas usando los comandos `ptime`, `cache.statistics` y `print-statistics -all`. Después de cada muestra almacenamos un checkpoint para poder volver a ese punto si se produce algún problema durante la simulación.

Para pasar a la siguiente muestra, debemos saltar los ciclos que nos interese. Durante ese tiempo, desactivamos las caches y la función que se ejecuta al cambiar entre modo usuario y modo protegido para acelerar la simulación. Después de saltar los ciclos almacenamos otro checkpoint. Nos interesa que la cache contenga información válida como si hubiéramos estado usándola durante todo el tiempo para que al iniciar la muestra no haya un exceso de fallos que no se corresponda con la ejecución normal del programa. Por lo tanto, iniciamos la cache y ejecutamos durante 100 millones de ciclos para calentarla. A continuación, volvemos a utilizar un bucle para tomar todas las estadísticas de la muestra. Repetiremos este proceso tantas veces como sea necesario para obtener todas las muestras.

C.3 Ejecución de las simulaciones

Cuando hay que ejecutar un gran número de simulaciones es importante automatizar el proceso para que resulte más cómodo y rápido. Nosotros hemos llevado a cabo las simulaciones en el cluster del departamento, en el cual podemos utilizar Condor [17], un sistema de gestión de cargas de trabajo. Añadimos a continuación un ejemplo de su uso para lanzar un par de simulaciones:

```
executable = ./simics
universe = vanilla
should_transfer_files = yes
when_to_transfer_output = on_exit_or_evict
environment = LM_LICENSE_FILE=/home/ortin/common/simics

# -------------------------------------------------------------------------

arguments = -stall -no-win runProc001ParsecBlackscholesNative00004k.simics
output = Proc001ParsecBlackscholesNative00004k.out
error = Proc001ParsecBlackscholesNative00004k.error
log = Proc001ParsecBlackscholesNative00004k.log
```

```
queue

# ----------------------------------------------------------------------

arguments = -stall -no-win runProc001ParsecBlackscholesNative00008k.simics
output = Proc001ParsecBlackscholesNative00008k.out
error = Proc001ParsecBlackscholesNative00008k.error
log = Proc001ParsecBlackscholesNative00008k.log

queue

...
```

Suponiendo que el script se llama `myCondorScript.txt`, para ejecutar las tareas programadas sólo tenemos que utilizar el comando:

```
condor_submit myCondorScript.txt
```

En nuestro caso, el uso de Condor estaba muy limitado, ya que nuestras simulaciones eran demasiado costosas en tiempo y recursos. La configuración de Condor en nuestro cluster no permitía colocar los procesos en cola e ir ejecutándolos poco a poco, ni garantizar que tendrían mayor prioridad que otros procesos. Por lo tanto, las simulaciones eran expulsadas cuando llevaban varios días ejecutándose y tenían que volver a empezar. Finalmente, fue necesario entrar en cada nodo del cluster e ir lanzando las simulaciones por medio de pequeños scripts controlando periódicamente la ocupación del cluster de forma manual.

C.4 Recopilación de resultados

Al terminar las simulaciones, hay que recopilar los resultados y presentarlos de manera que podamos analizarlos con comodidad. Debido al gran volumen de información obtenido, resulta impracticable obtener los datos necesarios a mano a partir de los ficheros de salida. Para representar los datos hemos elaborado gráficas utilizando gnuplot [1], así que al recoger los valores que nos interesaban de los ficheros, los hemos ordenado de manera que luego nos resultara cómodo utilizarlos directamente como ficheros de datos para dibujar las figuras. Hemos automatizado el proceso de recogida de datos utilizando scripts de shell. Mostraremos el script utilizado para recoger los datos de las trazas temporales en la entrada nativa, que es el más complejo. Como punto de partida, tenemos un fichero en el que aparecen las estadísticas de las diez muestras, separadas por las palabras "MUESTRA NUMERO X". Como salida, nos interesa tener un fichero por cada muestra de cada aplicación, con una línea por cada vez que se han tomado las estadísticas (cada diez millones de ciclos). En cada línea queremos que aparezcan el número de lecturas y escrituras de datos, número de fallos, instrucciones ejecutadas hasta el momento e instrucciones del último intervalo. Se incluyen comentarios explicativos sobre el propio código.

```
# Obtenemos los datos para todos los benchmrks de PARSEC
for DIRECTORY in ./parsec/*
do
    #Creamos un fichero distinto para cada muestra
    csplit -f "$DIRECTORY"/muestra -s "$DIRECTORY"/*native.out \
          /MUESTRA\ NUMERO/ /MUESTRA\ NUMERO/ /MUESTRA\ NUMERO/ \
          /MUESTRA\ NUMERO/ /MUESTRA\ NUMERO/ /MUESTRA\ NUMERO/ \
         /MUESTRA\ NUMERO/ /MUESTRA\ NUMERO/ /MUESTRA\ NUMERO/ \
         /MUESTRA\ NUMERO/

    INITIALINTERVAL=0
    ITER=0

    # Obtenemos los datos de cada muestra
    for FILENAME in "$DIRECTORY"/muestra*
    do
        ITER=`expr $ITER + 1`

        # Guardamos en un fichero distinto los valores de accesos,
        # fallos e instrucciones de cada iteración.
        grep "Data read transactions:" "$FILENAME" \
                 | sed 's/[a-zA-Z:]* //g' >"$FILENAME".read
        grep "Data write transactions:" "$FILENAME" \
                 | sed 's/[a-zA-Z:]* //g' >"$FILENAME".write
        grep "Data read misses:" "$FILENAME" \
                 | sed 's/[a-zA-Z:]* //g' >"$FILENAME".readmiss
        grep "Data write misses:" "$FILENAME" \
                 | sed 's/[a-zA-Z:]* //g' >"$FILENAME".writemiss
        grep "instructions executed" "$FILENAME" \
                 | awk '{print $1}' > aux.txt

        # Para las instrucciones sólo tenemos información de las
        # instrucciones ejecutadas hasta el momento, pero queremos
        # calcular también las instrucciones del último intervalo.
        awk '
            NR<2 { print int($1), int($1) }
            NR>=2 { print int($1), int($1)-orig }
            { orig= int($1) }
        ' aux.txt >"$FILENAME".numinstr

        rm aux.txt

        # Por último, unimos el contenido de todos los ficheros.
        pr -tmJ "$FILENAME".numTrans "$FILENAME".read "$FILENAME".write \
                   "$FILENAME".readmiss "$FILENAME".writemiss \
                   "$FILENAME".numinstr >"$FILENAME".allstats \

    done
done
```

A partir de los ficheros obtenidos podemos crear las gráficas que nos permitirán analizar los resultados y obtener conclusiones. Los scripts para recopilar los resultados de otras simulaciones se han elaborado de manera similar y no se incluyen en este documento porque no aportan ya información adicional.

Anexo D
Resultados de la caracterización de PARSEC

Este anexo recoge un análisis detallado de las estadísticas recogidas a partir de los experimentos y las conclusiones a las que se ha llegado. Además, se incluye una selección de las entradas que se deberán utilizar para simular cada una de las aplicaciones de la suite Parsec en un tiempo razonable y obteniendo resultados representativos.

D.1 Impacto del tamaño de las entradas en la jerarquía de memoria

En esta sección se presentan los resultados de los experimentos descritos en el capítulo 4, cuya finalidad es analizar el impacto del tamaño de las entradas de las aplicaciones de PARSEC en la jerarquía de memoria. Se comienza presentando los resultados correspondientes al instruction mix y al footprint, que verifican la idea de que las entradas de menor tamaño son versiones reducidas de la entrada nativa y que esta última supone un mayor uso de recursos. A continuación, se introducen los fallos de TLB, donde aparecen los primeros indicios de que no necesariamente las entradas más grandes son las que más presionarán a la jerarquía de memoria. Por último, se muestran las tasas de fallos y trazas temporales, en las que vemos claramente que las entradas nativas no se diferencian excesivamente del resto y que no siempre son las que mayores tasas de fallos presentan. Esto nos lleva a pensar que será posible obtener resultados representativos de una ejecución con entrada nativa sin necesidad de realizar simulaciones tan costosas.

D.1.1 Instruction mix

El porcentaje de instrucciones de lectura y escritura que se ejecutan para cada entrada de todas las aplicaciones de PARSEC aparece representado en la figura 5.1. Se ve claramente que, en todos los casos, la proporción se mantiene prácticamente igual para todas las entradas de una misma aplicación. Encontramos una excepción en `dedup`, cuya entrada de tamaño grande se diferencia bastante del resto. Esto lo veremos también reflejado más adelante en otras métricas y lo explicaremos más en profundidad en la sección D.1.4.

En la figura D.1 aparece el número total de instrucciones que se ejecutan con cada una de las entradas de todas las aplicaciones, obtenido de la misma ejecución que los datos de la figura 5.1. Para representar los resultados con mayor claridad, se ha utilizado un eje logarítmico. Se puede comprobar que el número de instrucciones va aumentando según se incrementa el tamaño de la

entrada, especialmente al pasar de la entrada grande a la nativa. El único caso en el que no se cumple es `facesim`, en el que las entradas pequeña, mediana y grande tienen el mismo número de instrucciones. Esto se debe a que las tres entradas son idénticas, ya que el escalado supondría reducir el tamaño de la representación de la cara (recordamos que este benchmark se ocupa de la simulación del movimiento de una cara) y eso podría crear inestabilidad numérica [10]. El número de instrucciones ejecutadas junto con el instruction mix nos lleva a pensar que, efectivamente, las entradas más pequeñas son una aproximación reducida de las entradas más grandes.

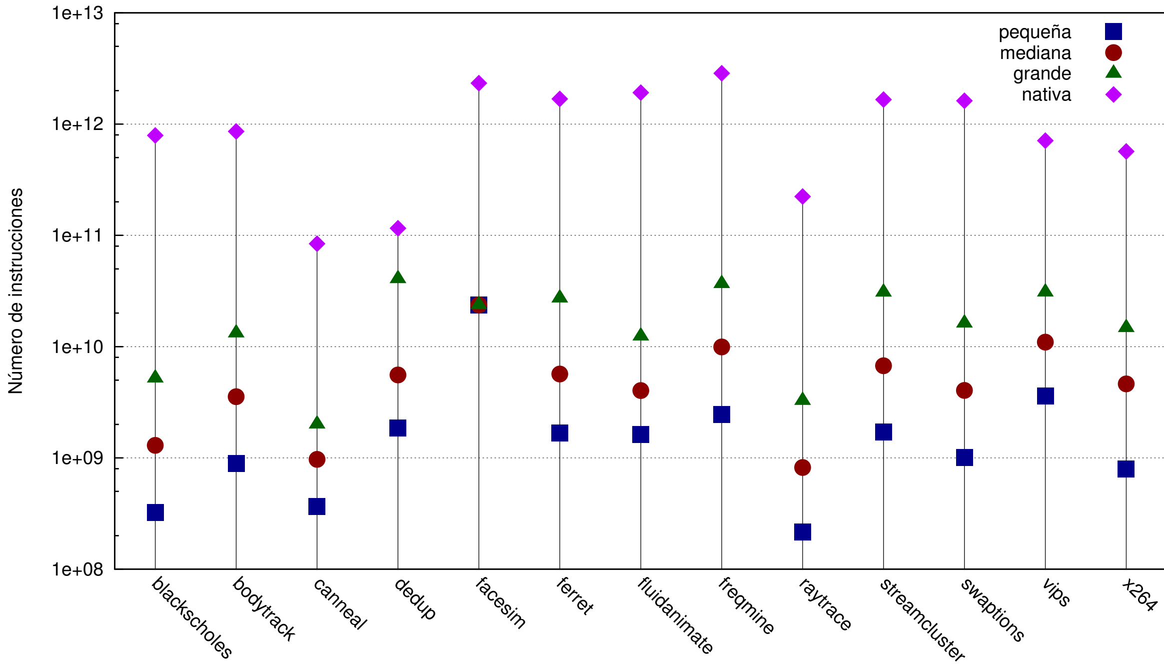


Figura D.1: Número de instrucciones de las aplicaciones de PARSEC con cada una de sus entradas

D.1.2 Footprint

En la figura 5.2 mostrábamos la cantidad de memoria a la que acceden las aplicaciones durante la región de interés. Veíamos que la cantidad de memoria accedida para instrucciones se mantiene constante, mientras que la de datos aumenta con el tamaño de la entrada a excepción de las aplicaciones `facesim` y `vips`. En la figura D.2 aparece representado el número medio de accesos a cada página de memoria junto con la desviación típica. En todos los casos, el número de accesos por página, tanto de instrucciones como de datos, es mayor para las entradas más grandes. Al igual que en la sección anterior, esto nos sigue confirmando que las entradas mayores realizarán un uso más intensivo de la memoria.

En las figuras 4.1, D.3 y D.4 aparece el footprint del 50 %, 90 % y 100 % de los accesos a memoria para datos, tal y como había sido explicado en la sección 4.2. Todas las aplicaciones presentan mucha localidad espacial, ya que muchos de los accesos se concentran en un conjunto pequeño de las páginas. Además, hay muchos casos en los que la localidad es claramente mayor para las aplicaciones nativas, ya que la cantidad de memoria correspondiente al 50 % o incluso al 90 % de los accesos se mantiene baja, mientras que el footprint total aumenta (`blackscholes`, `dedup`, `ferret`, `fuidanimate`, `frqmine`, `raytrace`, `swaptions`, `vips` y `x264`).

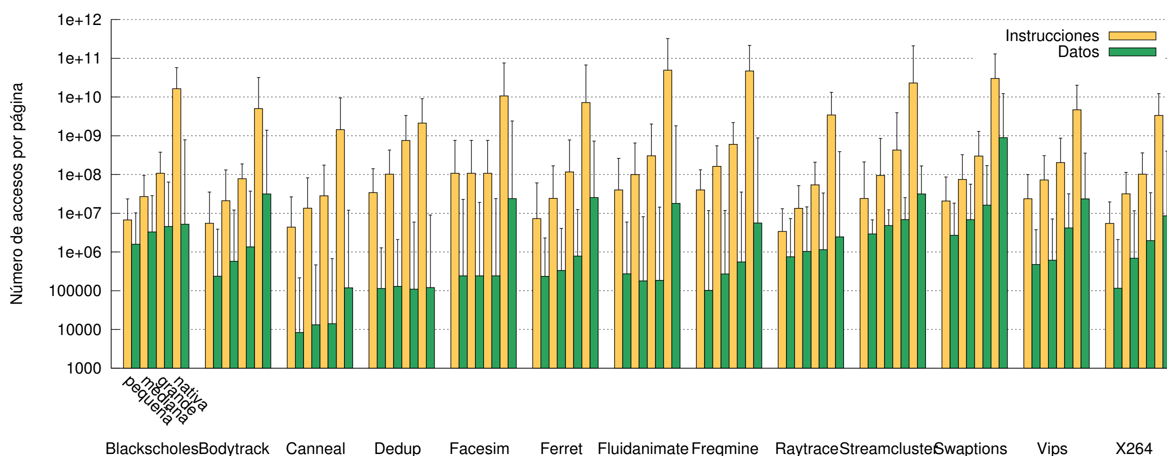


Figura D.2: Número medio de accesos a cada página de memoria de las aplicaciones de PARSEC, diferenciando entre accesos para instrucciones y para datos.

D.1.3 Fallos de TLB

En la figura 5.3 presentábamos los fallos que se producen en los TLB de datos de los dos niveles. Una descripción de la estructura del TLB y la recopilación de estas estadísticas puede consultarse en la sección 4.3. Las escrituras van directamente al TLB de nivel superior, pero las lecturas pasan por ambos niveles, así que podemos ver los accesos que fallan en L0 pero aciertan en el siguiente nivel y los que fallan en ambos niveles. Los datos que representamos son los fallos cada mil instrucciones (*misses per kiloinstruction* o MPKI). Esto nos da una idea del número de fallos independientemente del tamaño del programa, lo que nos permite comparar unos con otros cómodamente.

En este caso vemos que ya no se repite un patrón tan claramente como en la sección anterior, el número de fallos de TLB no va aumentando progresivamente con el tamaño de la entrada. En algunos casos, el número de fallos decrece (`blackscholes`, `bodytrack`, `ferret`, `raytrace` y `vips`) y en otros casos se aprecia una forma de "U" (`canneal`, `dedup`, `fluidanimate`, `swaptions` y `x264`). Por lo tanto, ya no parece que se cumpla la suposición de que el escalado de las entradas implica que las más grandes sean más fieles a una ejecución real con mayor número de fallos.

D.1.4 Tasas de fallos en cache y trazas temporales

En esta sección se van a presentar y analizar los resultados del estudio del comportamiento de las aplicaciones de PARSEC sobre la cache de primer nivel del procesador. En concreto, para cada benchmark se tendrán en cuenta las siguientes estadísticas:

- Fallos de lectura y escritura con diferentes tamaños de cache en una arquitectura Intel, con política *write-allocate* (cuando se produce un fallo de escritura, el bloque correspondiente se trae a la cache) y *copy-back* (las escrituras se realizan sobre el bloque en cache y se copian a memoria principal cuando este se reemplaza). Estos resultados se han obtenido utilizando VALGRIND, tal y como se explica en la sección 4.4.

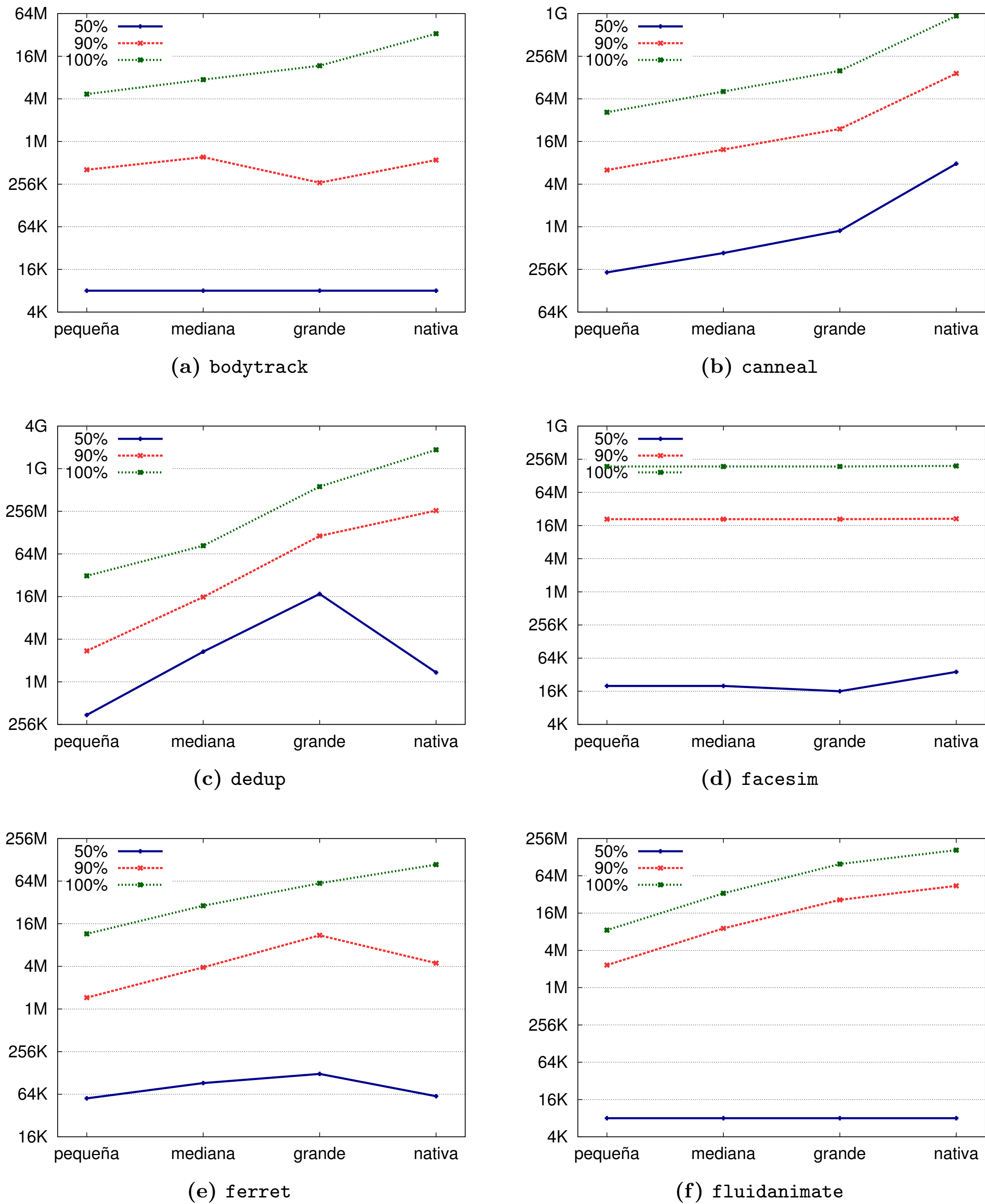


Figura D.3: 50 %, 90 % y 100 % del footprint de las aplicaciones de PARSEC (parte 1)

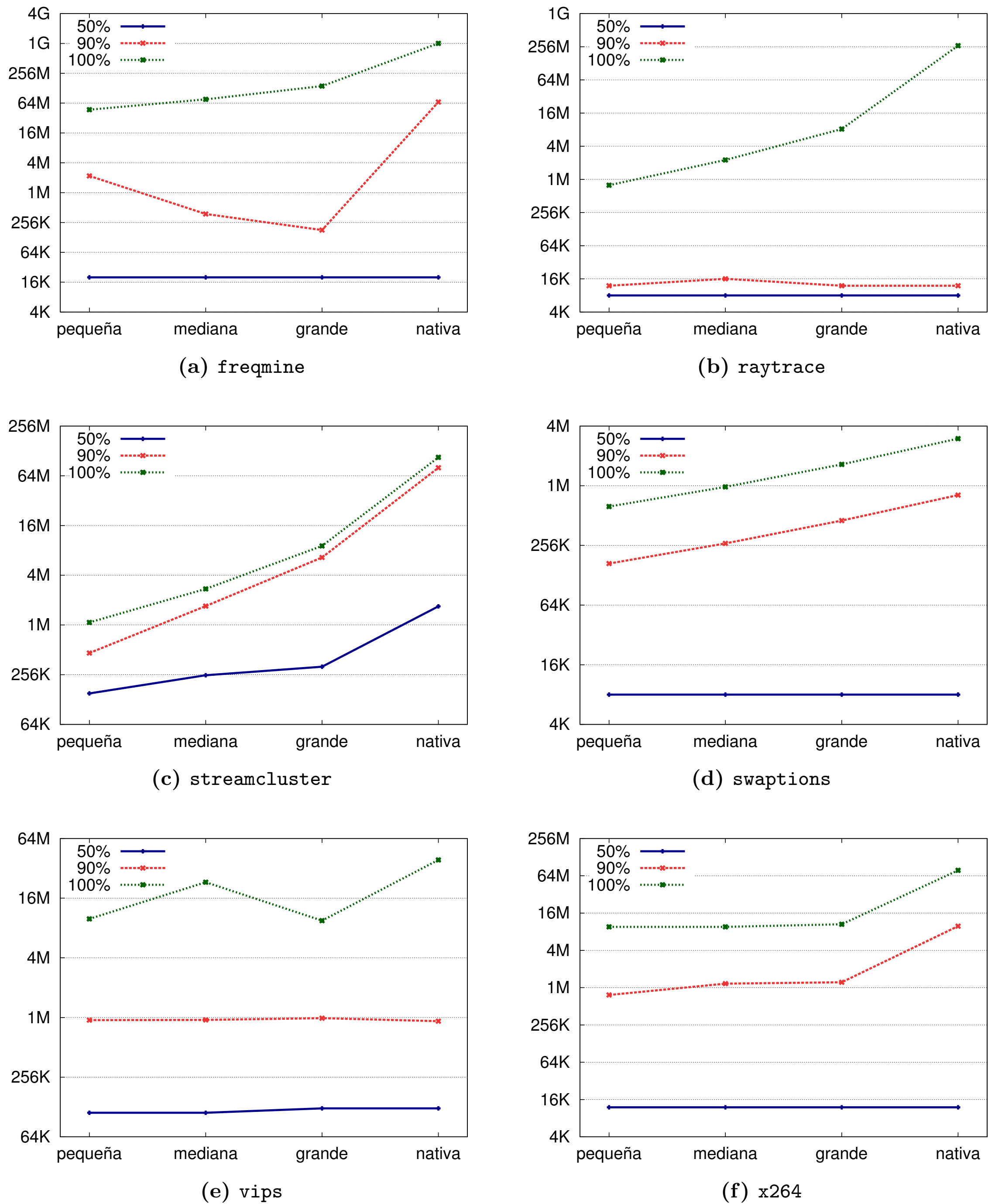


Figura D.4: 50 %, 90 % y 100 % del footprint de las aplicaciones de PARSEC (parte 2)

- Fallos de lectura y escritura con diferentes tamaños de cache en una arquitectura Sparc. Para este apartado se han tomado las medidas tanto para una política *non-write-allocate* y *write-through* (las escrituras se realizan directamente en memoria) como para una write-allocate y copy-back. En el caso de la política non-write-allocate y write-thorugh, se han tenido en cuenta tanto las instrucciones de usuario como las de sistema, ya que el no tener que distinguir entre las dos acelera el proceso de simulación. De todas formas, las instrucciones de usuario son prácticamente el total de las instrucciones ejecutadas. No se dispone de resultados para algunas simulaciones con entrada nativa debido al elevado tiempo de simulación (más de un mes en varios casos). Estos datos se han obtenido mediante simulación con Simics, como se describe en la sección 4.5.

- Traza temporal de los fallos con política write-allocate utilizando una cache de 64 KB. Se ha seguido el procedimiento explicado en la sección 4.5.

En todos los casos se presentan los fallos en MPKI (misses per kiloinstruction), al igual que en la sección D.1.3. En la sección 5.2.4 se incluye una explicación de cómo aparecen representados los datos en cada una de las gráficas utilizadas a lo largo de los siguientes apartados.

Blackscholes

En las figuras 5.4 y 5.5 veíamos los fallos por cada mil instrucciones para cada una de las entradas del benchmark `blackscholes`, variando el tamaño de la cache en Intel y Sparc. En esta aplicación el número de fallos de las entradas más pequeñas disminuye bruscamente a partir del punto en que las estructuras principales caben en la cache, indicando que la entrada nativa sí que genera más fallos en cache y estresa más la jerarquía de memoria.

Analizando la traza temporal para todas las entradas que presentábamos en las figuras 5.6 y 5.7, vemos que el número de fallos se mantiene prácticamente constante en todos los casos. En la entrada nativa, a excepción de una anomalía alrededor del ciclo 4500, aparecen picos de fallos cada 750 millones de ciclos.

Bodytrack

Fijándonos en las figuras D.5 y D.6 vemos que con esta aplicación ya no se cumple tan claramente que las entradas más grandes supongan un número mayor de fallos en cache. En el caso de la arquitectura Intel, las entradas mediana y grande tienen siempre más fallos. Si nos centramos en los resultados para la arquitectura Sparc, la entrada grande presenta más fallos que la nativa en varios casos (aunque no se disponga de todos los datos para el protocolo write-allocate recordamos que consideramos que los resultados seguirán el mismo patrón que con non-write-allocate, tal y como hemos explicado en el apartado anterior). A parte de eso, vemos que con caches de 256 KB o mayores el número de fallos es muy pequeño, así que no podremos realizar estudios demasiado exhaustivos sobre la jerarquía.

La traza temporal (figuras D.7 y D.8) nos aporta una información muy valiosa en este caso. Vemos claramente que hay un patrón que se repite una vez en la entrada pequeña, dos en la mediana, cuatro en la grande y muchas veces en la nativa. Además, aunque este patrón tiene siempre la misma forma, no ocupa en todos los casos el mismo número de ciclos.

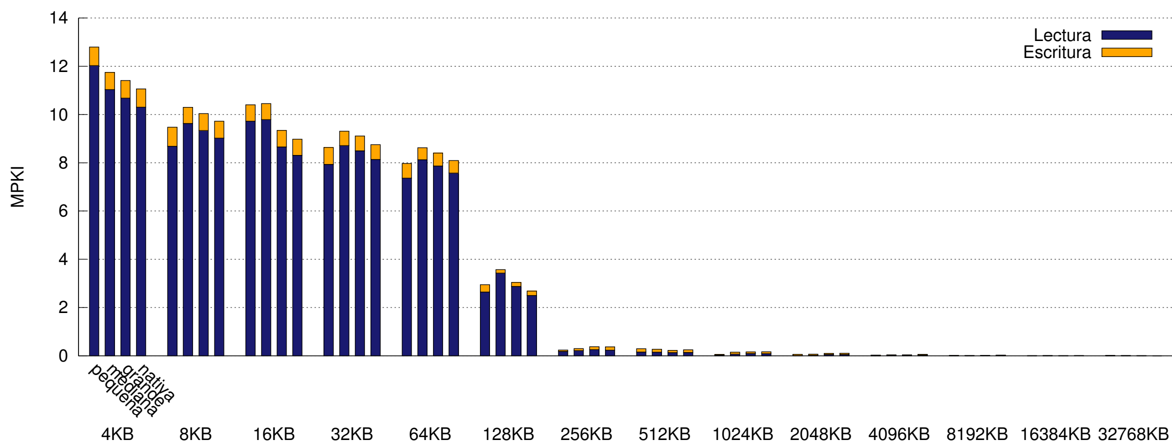
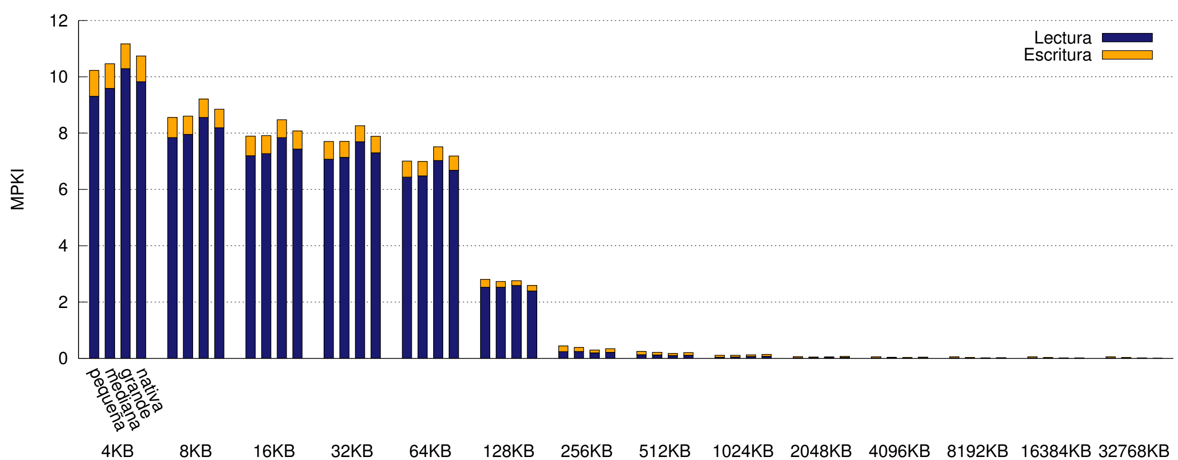
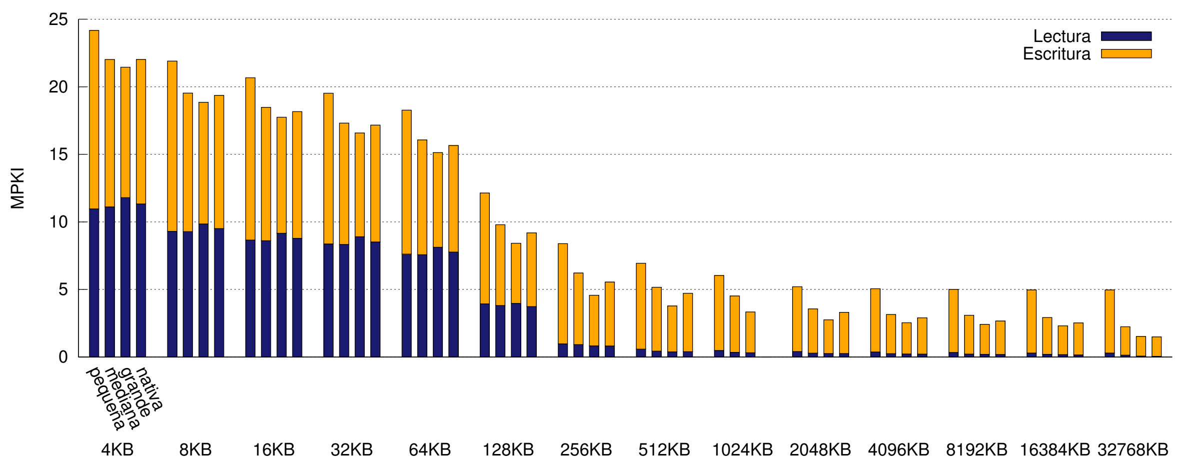


Figura D.5: Fallos por cada mil instrucciones en la cache de datos para `bodytrack` ejecutado en Intel. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

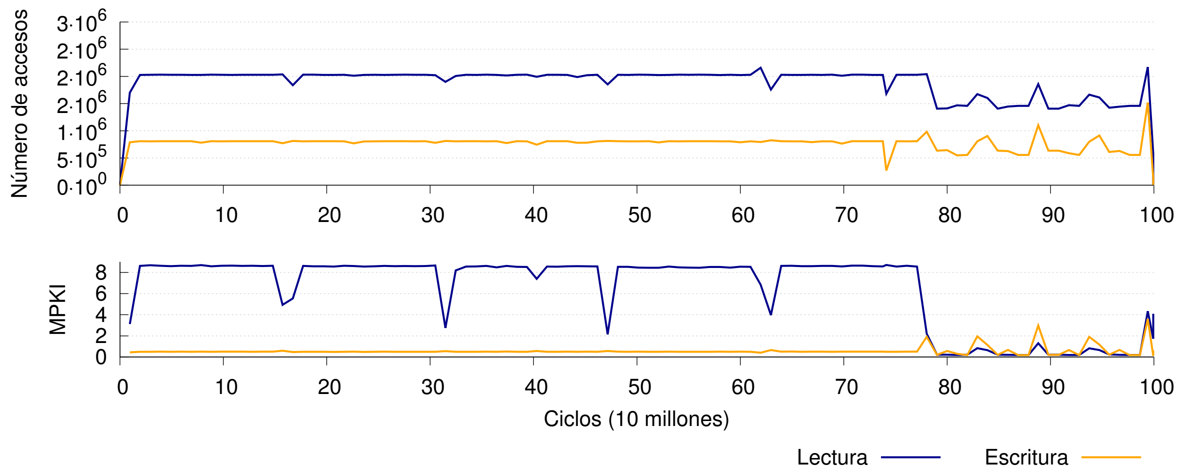


(a) write-allocate y copy-back, sólo instrucciones de usuario

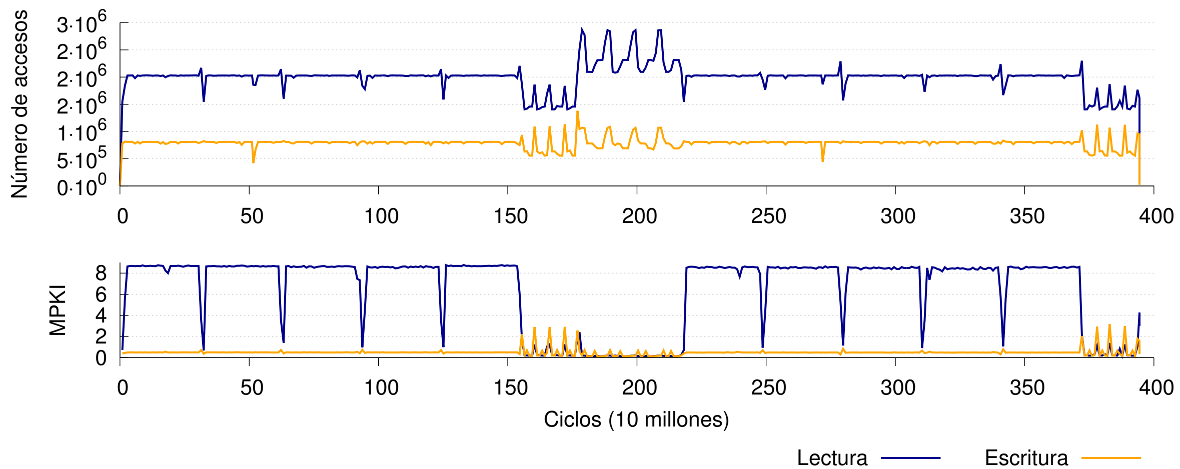


(b) non-write-allocate y write-through, instrucciones de usuario y sistema

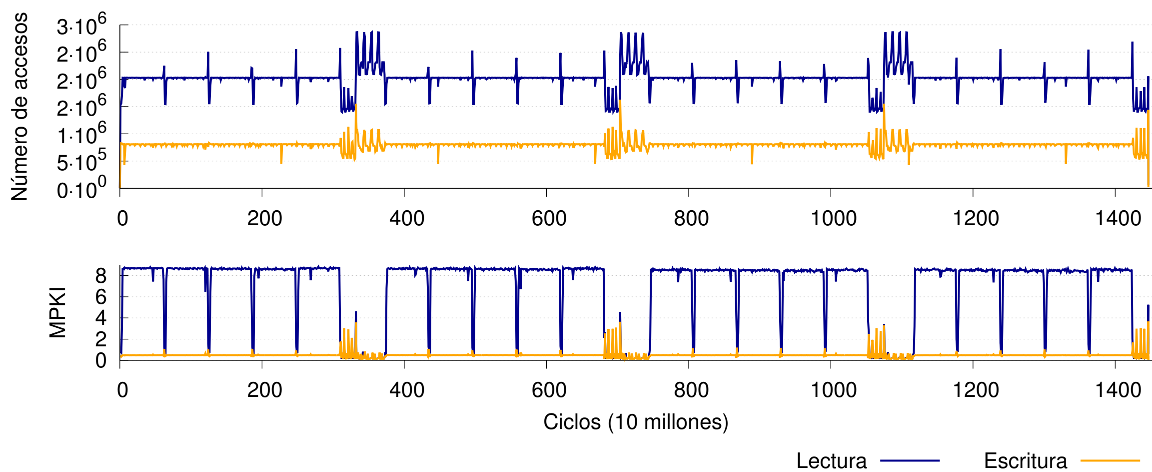
Figura D.6: Fallos por cada mil instrucciones en la cache de datos para `bodytrack` ejecutado en Sparc.



(a) pequeña



(b) mediana



(c) grande

Figura D.7: Traza temporal de fallos en cache para `bodytrack` con entradas pequeña, mediana y grande, ejecutado en Sparc. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

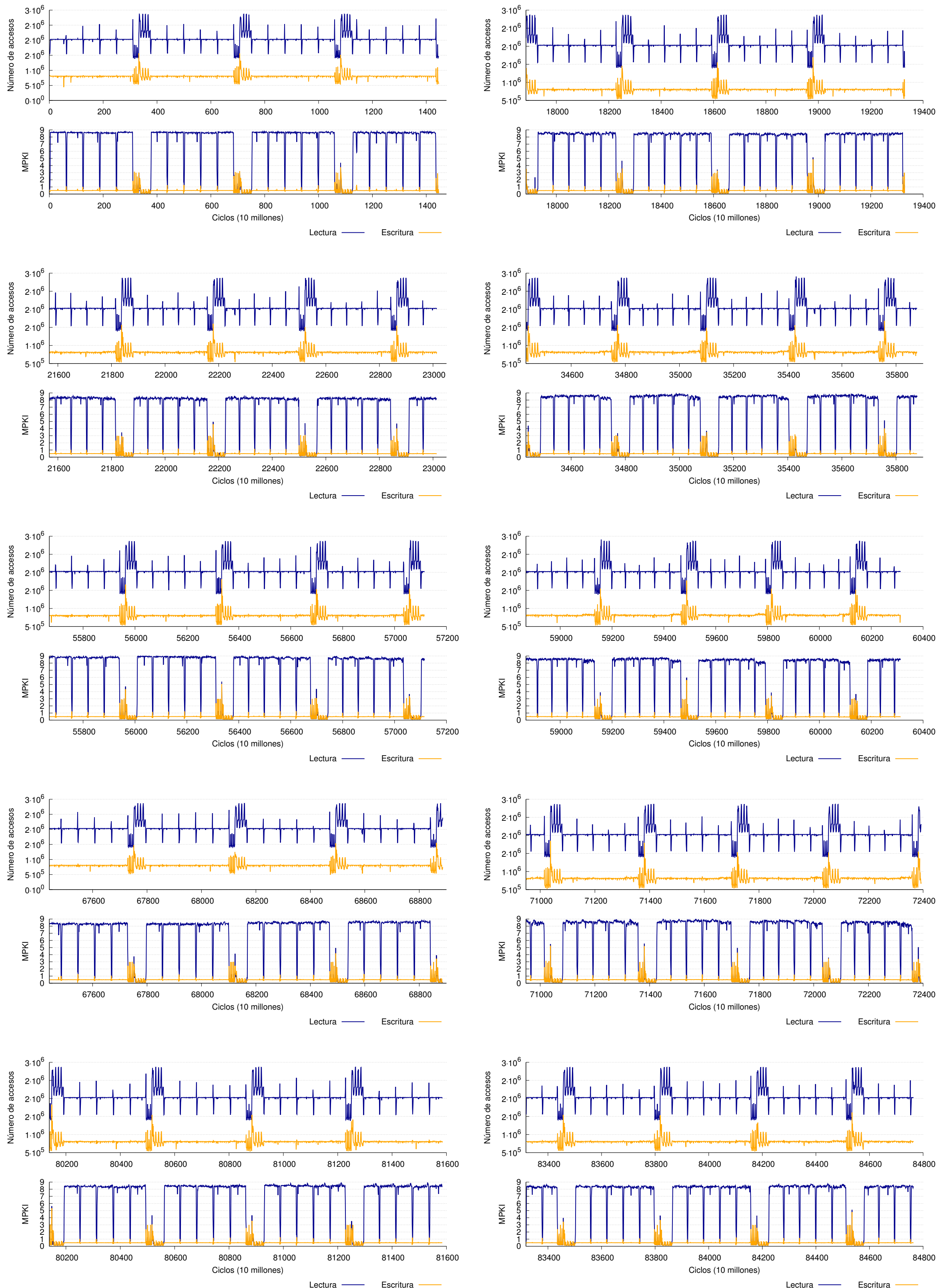


Figura D.8: Traza temporal de fallos en cache para `bodytrack` con entrada nativa, ejecutado en Sparc. Aparecen diez muestras tomadas al azar del total de la ejecución. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

Canneal

En la figura D.9 vemos los fallos en cache para una arquitectura Intel. Con caches menores se ven más fallos en la entrada pequeña, y con caches mayores, en la entrada nativa. En Sparc (figura D.10) se sigue generalmente la pauta de que entradas más grandes presentan mayor número de fallos.

Si nos centramos ahora en las trazas temporales (figuras D.11 y D.12), vemos que no se detectan patrones ni elementos de especial interés. En este caso, las simulaciones de la entrada nativa se han realizado contabilizando tanto las instrucciones de sistema como las de usuario, debido a que la simulación resulta demasiado lenta si se tiene que activar y desactivar la cache al cambiar entre modo de usuario y de sistema. De todas formas, se ha comprobado que las instrucciones de sistema suponen un porcentaje muy pequeño del total de la ejecución. Se ve que el número de fallos es muy uniforme en todos los casos. En la ejecución nativa el número está en torno a los 25 fallos por cada mil instrucciones, y a partir de la cuarta muestra hay un pequeño escalón y los fallos se estabilizan en algo más de 25. Además, cada 750 millones de instrucciones aproximadamente hay un punto con menor número de fallos de lectura.

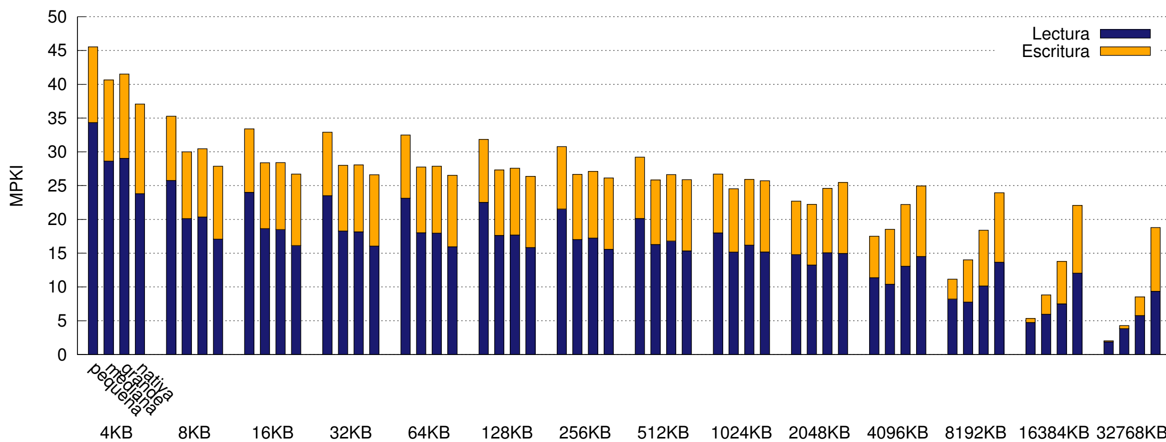
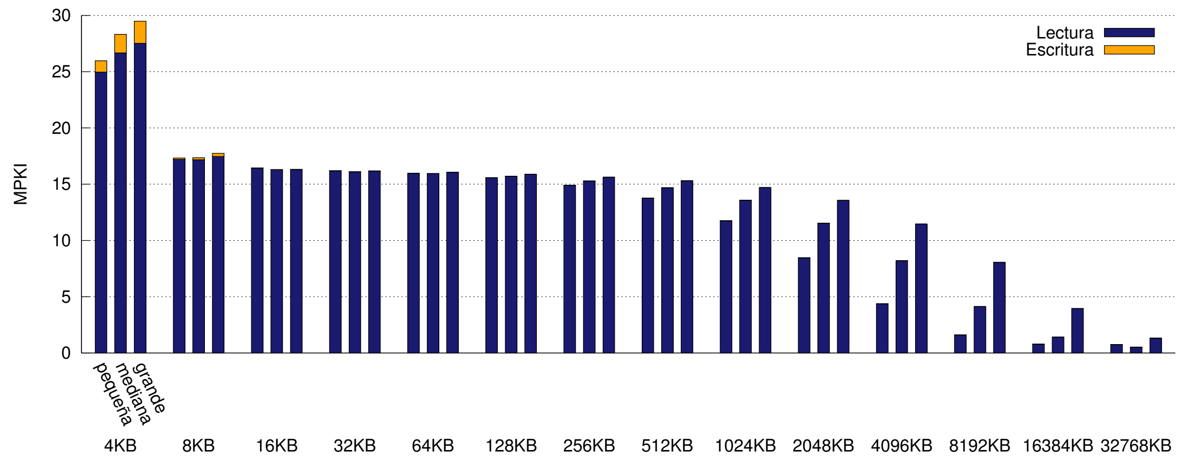
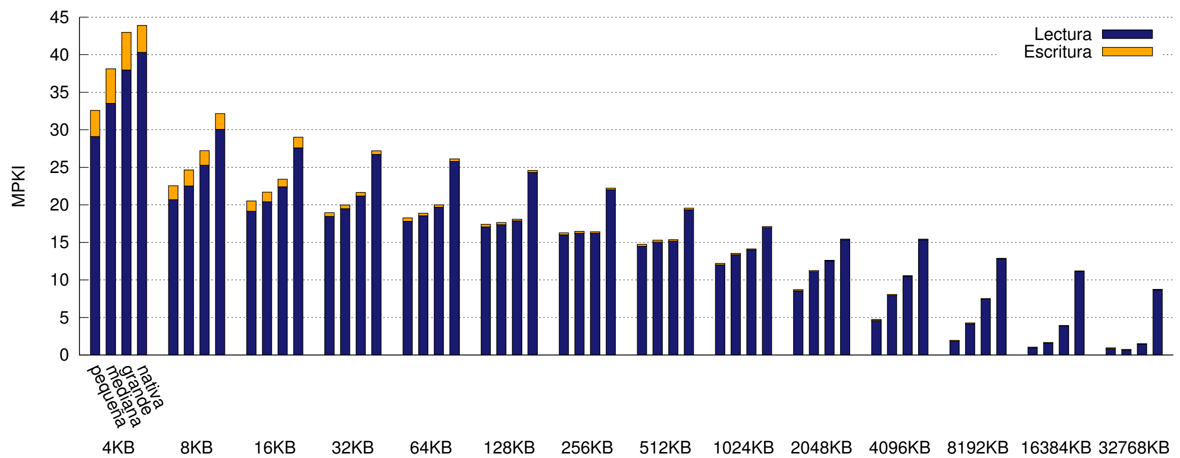


Figura D.9: Fallos por cada mil instrucciones en la cache de datos para `canneal` ejecutado en Intel. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

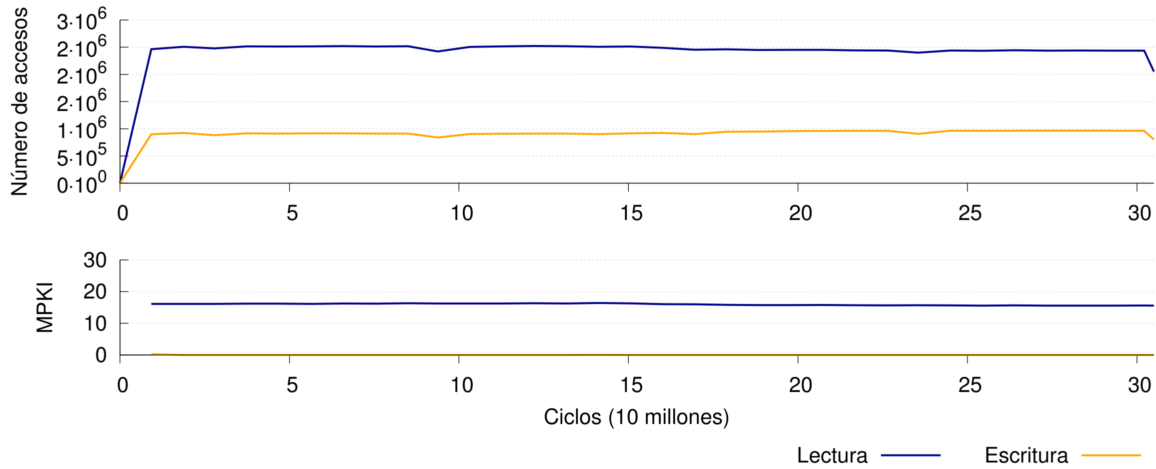


(a) write-allocate y copy-back, sólo instrucciones de usuario

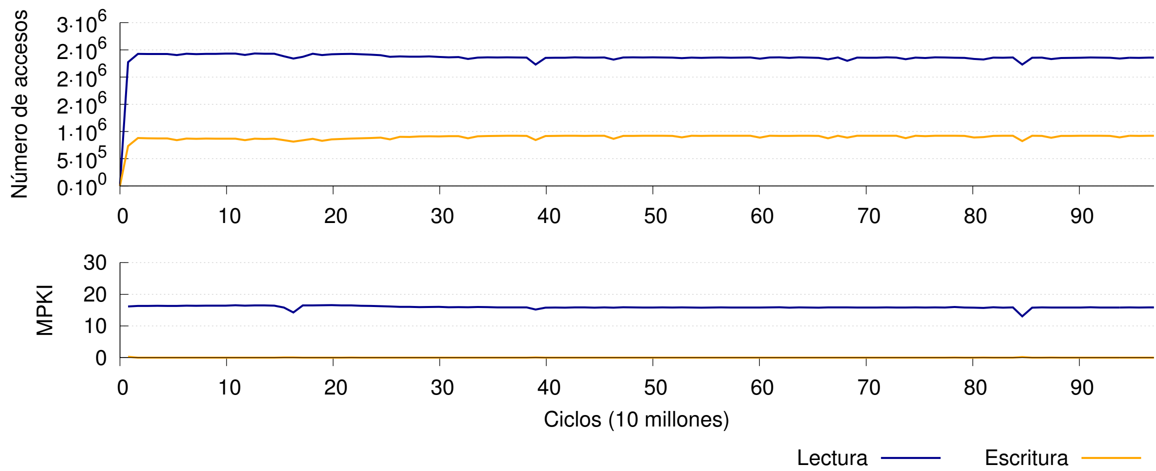


(b) non-write-allocate y write-through, instrucciones de usuario y sistema

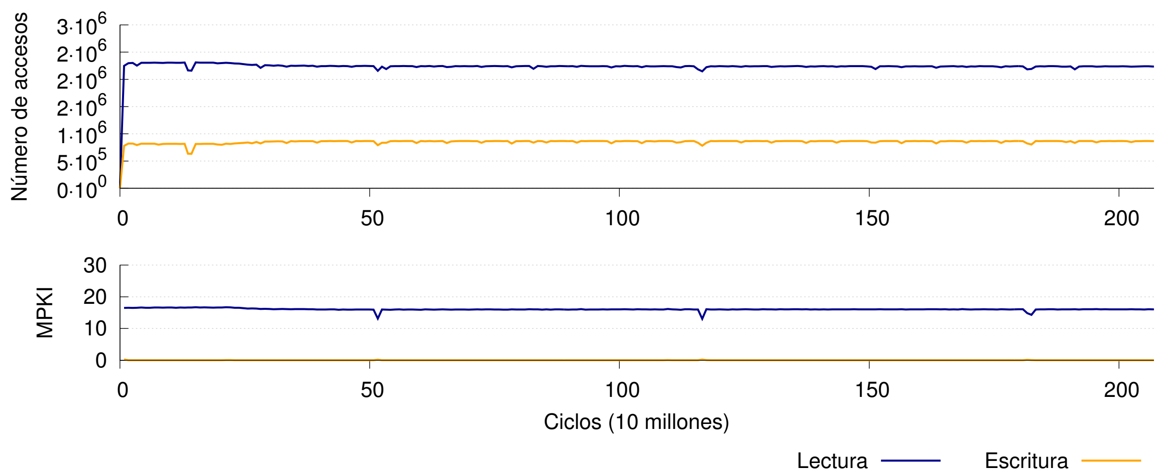
Figura D.10: Fallos por cada mil instrucciones en la cache de datos para `canneal` ejecutado en Sparc.



(a) pequeña



(b) mediana



(c) grande

Figura D.11: Traza temporal de fallos en cache para `canneal` con entradas pequeña, mediana y grande, ejecutado en Sparc. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

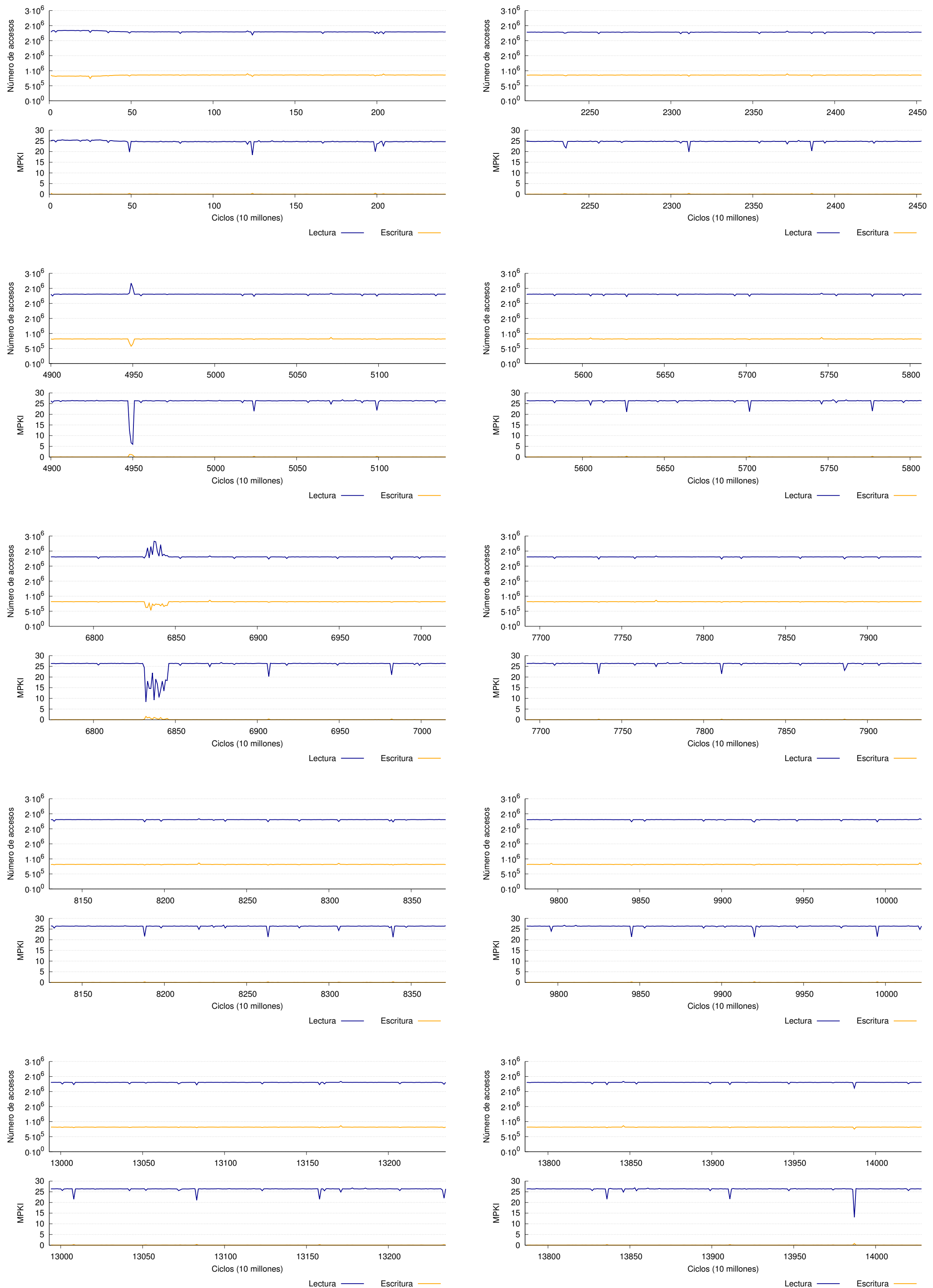


Figura D.12: Traza temporal de fallos en cache para `canneal` con entrada nativa, ejecutado en Sparc. Aparecen diez muestras tomadas al azar del total de la ejecución. Se utiliza una política write-allocate y copy-back.

Dedup

En las figuras D.13 y D.14 vemos que la entrada grande tiene, en muchos casos, menos fallos que el resto. Ya habíamos detectado este comportamiento diferente en el instruction mix en la sección D.1.1. En este benchmark, la ejecución se divide en cinco etapas, siendo paralelas las tres centrales (*pipeline parallelism*). El problema es que el pipeline está muy desbalanceado y en ocasiones se pasa demasiado tiempo en una etapa generando un cuello de botella. En [34] se estudia este problema y se plantean soluciones para mejorar este tipo de algoritmos.

En las figuras D.15 y D.16 se presenta la traza temporal de los fallos en cache. En este caso, la entrada nativa tiene únicamente el triple de instrucciones que la entrada grande, así que no se han realizado diez muestras en puntos aleatorios sino que se ha ejecutado todo el programa. El resultado se presenta dividiendo la traza en tres partes para que pueda estudiarse más claramente y para que la escala en el eje x sea igual a la utilizada para la entrada grande, como en el resto de aplicaciones.

Analizando las trazas de la figura D.15 vemos que en todos los casos podemos distinguir al menos dos partes. Mientras que para las entradas pequeña y mediana se aprecia un aumento de los accesos y fallos de lectura durante la segunda parte, en la entrada grande el número de fallos sube durante unos pocos ciclos pero luego se mantiene bajo hasta final de la ejecución. Esta diferencia es la que se refleja también en las figuras D.13 y D.14, y veíamos ya en el instruction mix (figura 5.1), y se debe a que el pipeline está desbalanceado. En la entrada nativa (figura D.16) se aprecia claramente un patrón, aunque distinguimos también tres partes que lo modifican ligeramente: la primera hasta las 3500 instrucciones aproximadamente, la segunda hasta 5300 y la última hasta el final de la ejecución.

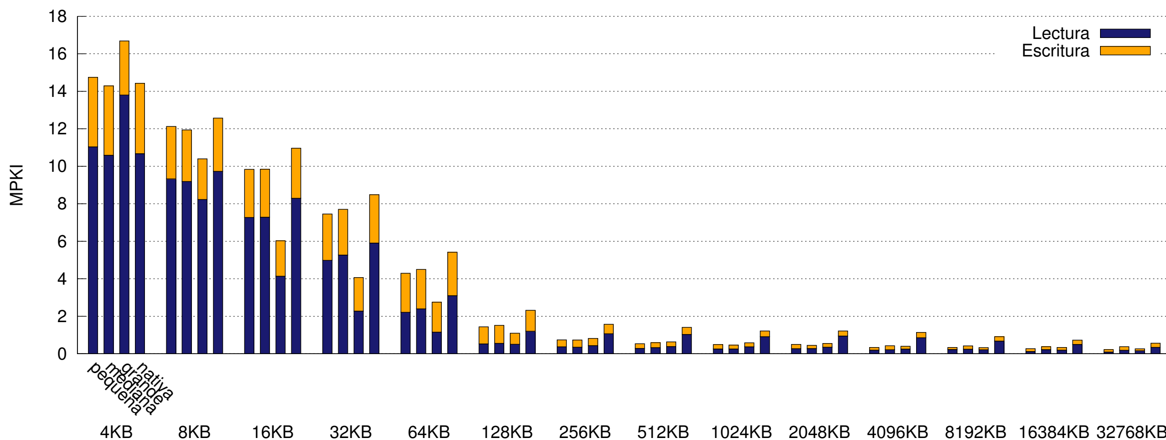
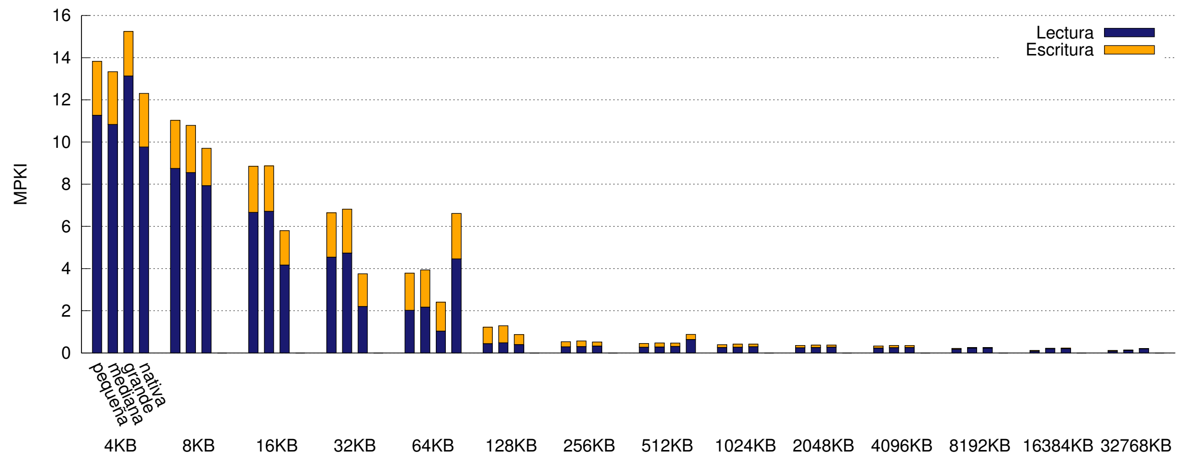
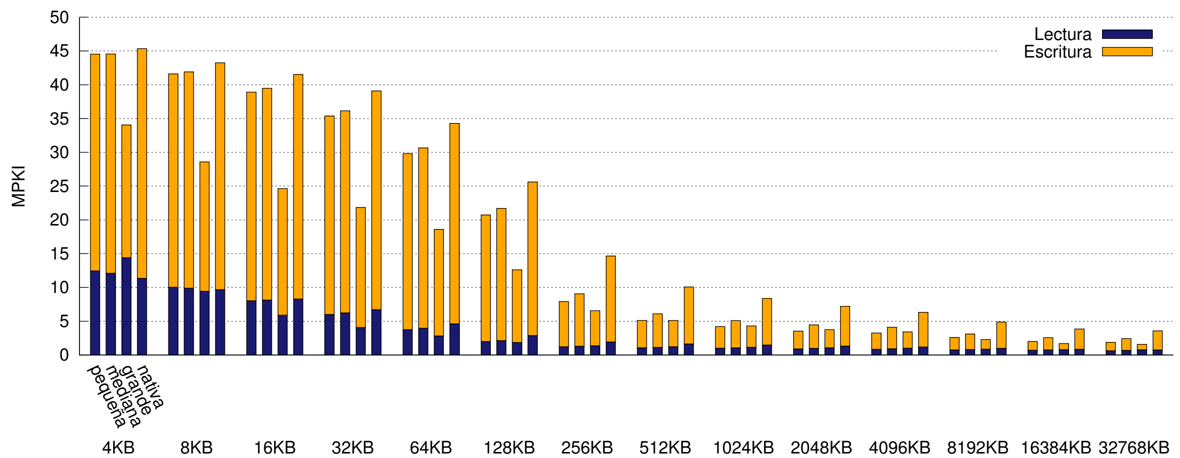


Figura D.13: Fallos por cada mil instrucciones en la cache de datos para `dedup` ejecutado en Intel. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

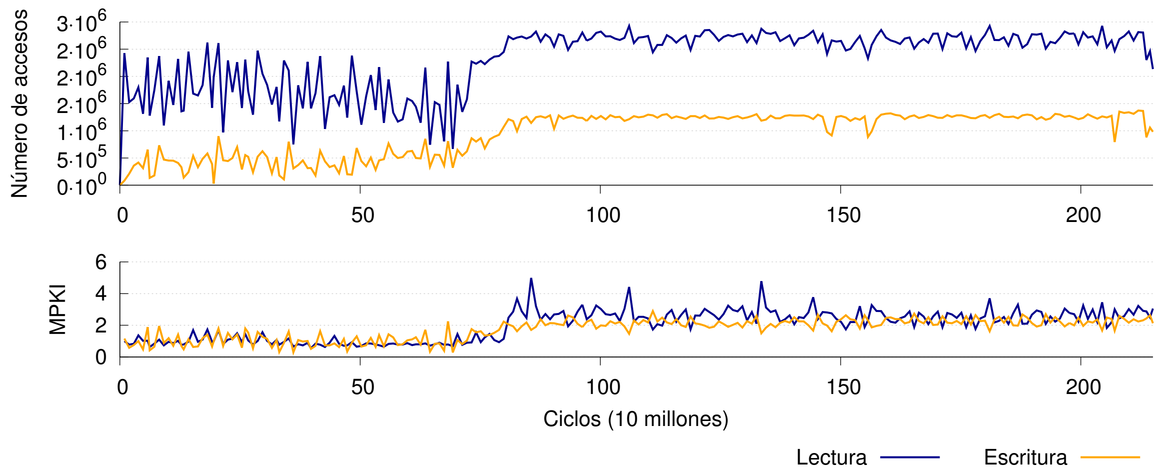


(a) write-allocate y copy-back, sólo instrucciones de usuario

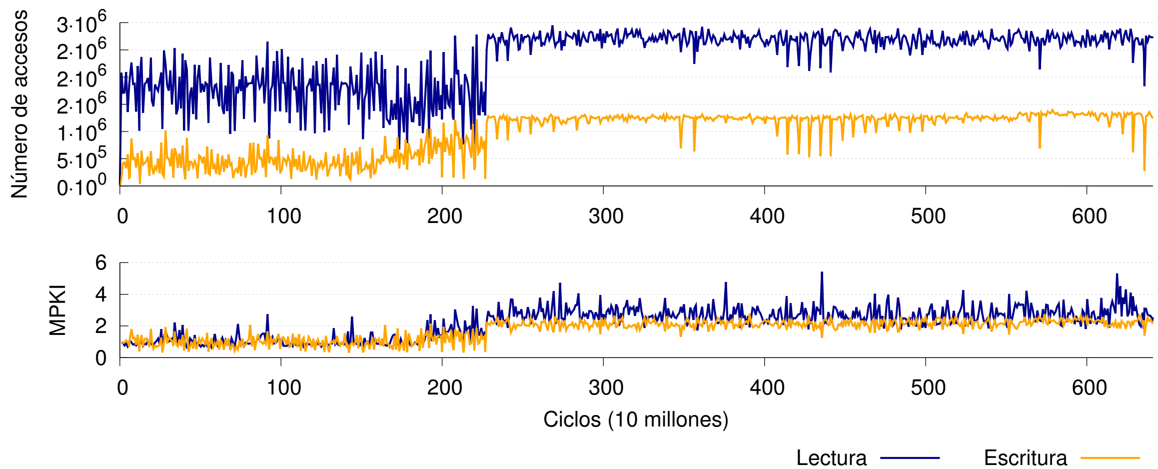


(b) non-write-allocate y write-through, instrucciones de usuario y sistema

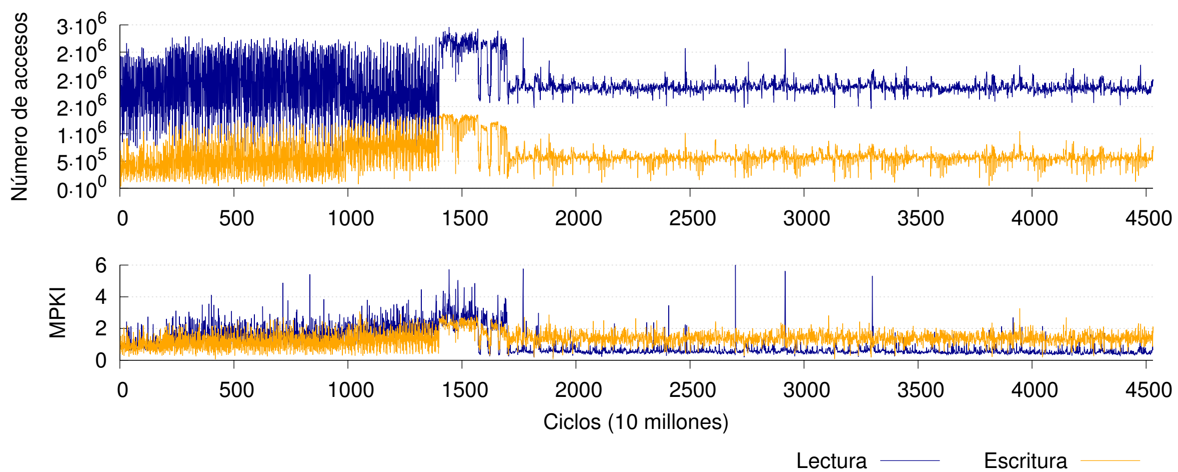
Figura D.14: Fallos por cada mil instrucciones en la cache de datos para `dedup` ejecutado en Sparc.



(a) pequeña



(b) mediana



(c) grande

Figura D.15: Traza temporal de fallos en cache para `dedup` con entradas pequeña, mediana y grande, ejecutado en Sparc. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

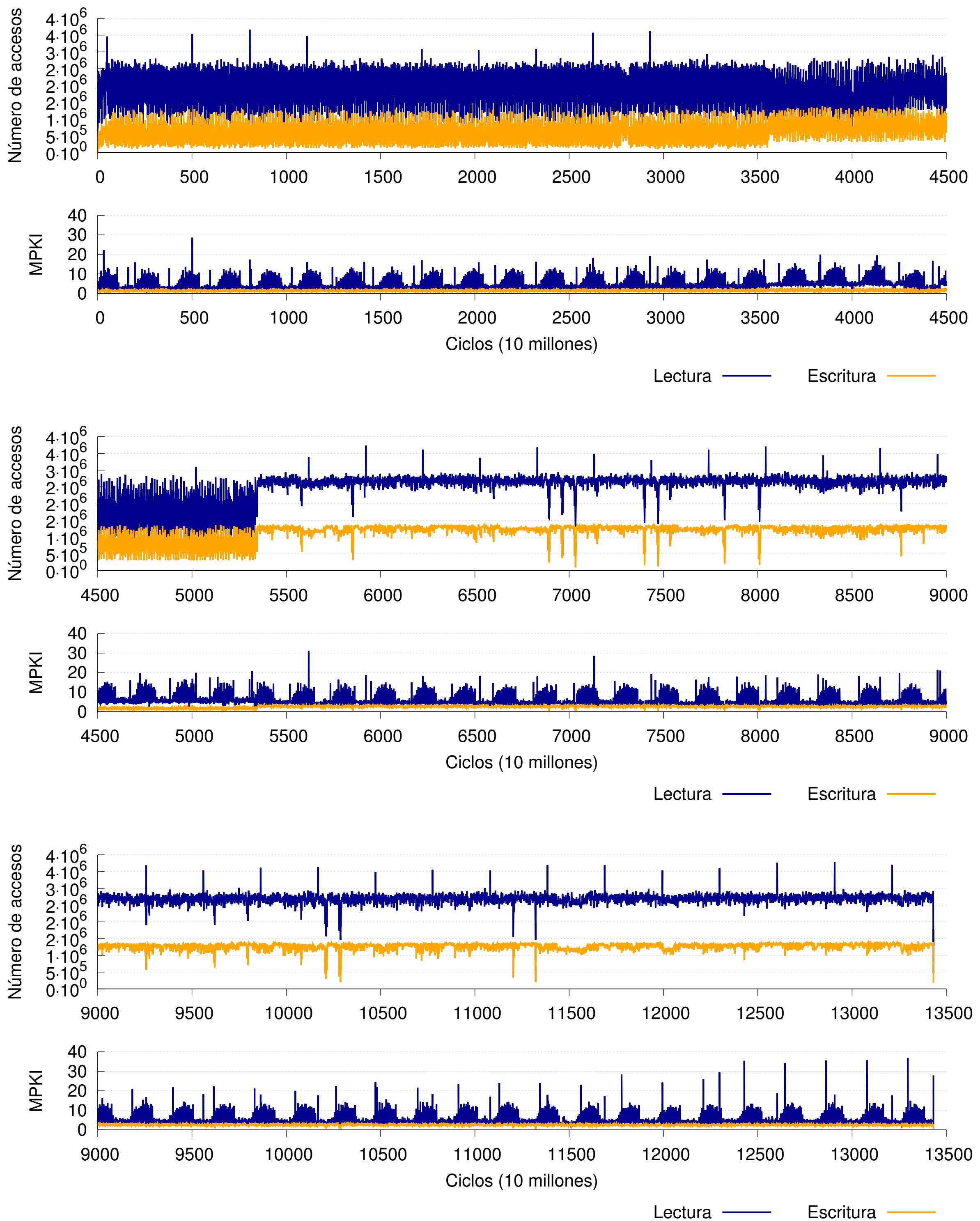


Figura D.16: Traza temporal de fallos en cache para `dedup` con entrada nativa, ejecutado en Sparc. Aparecen diez muestras tomadas al azar del total de la ejecución. Se utiliza una política write-allocate y copy-back.

Facesim

En las figuras D.17 y D.18 podemos comprobar que no hay apenas ninguna diferencia entre los fallos de todas las entradas, aunque modifiquemos el tamaño de la cache. Recordamos también que, en este caso, las entradas pequeña, mediana y grande son exactamente iguales, como ya se explicó en la sección D.1.1.

La traza temporal resulta en este caso muy útil. Para las entradas pequeña, mediana y grande (figura D.19) se aprecia una forma bastante característica. En la entrada nativa (figura D.20), esta forma exactamente igual se repite durante toda la ejecución. Por lo tanto, vemos claramente un patrón que aparece una vez en las entradas pequeña, mediana y grande y varias veces en la entrada nativa.

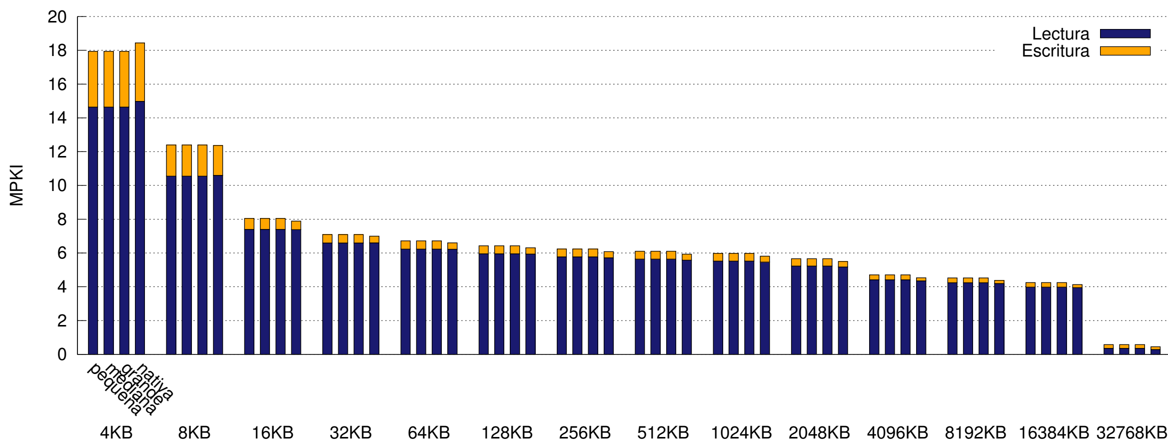
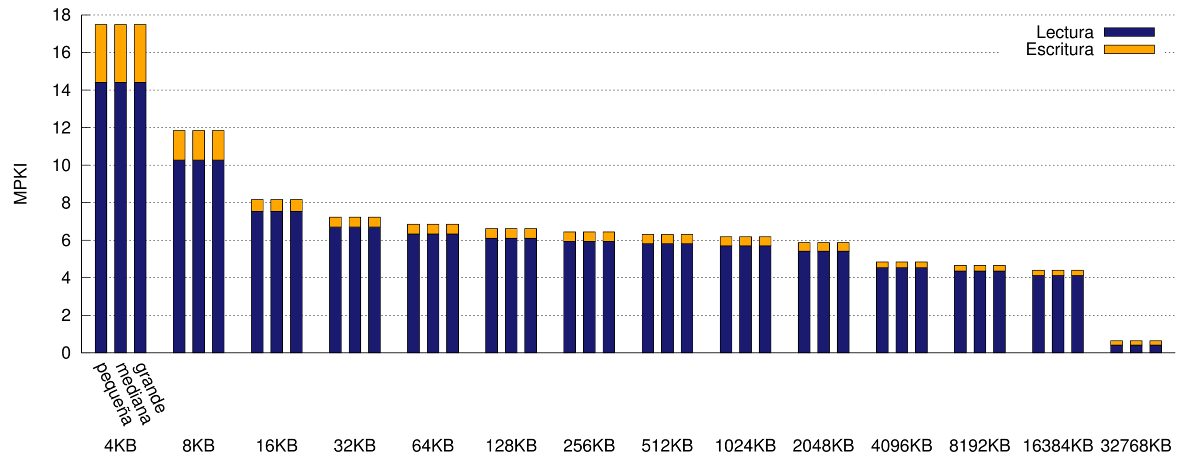
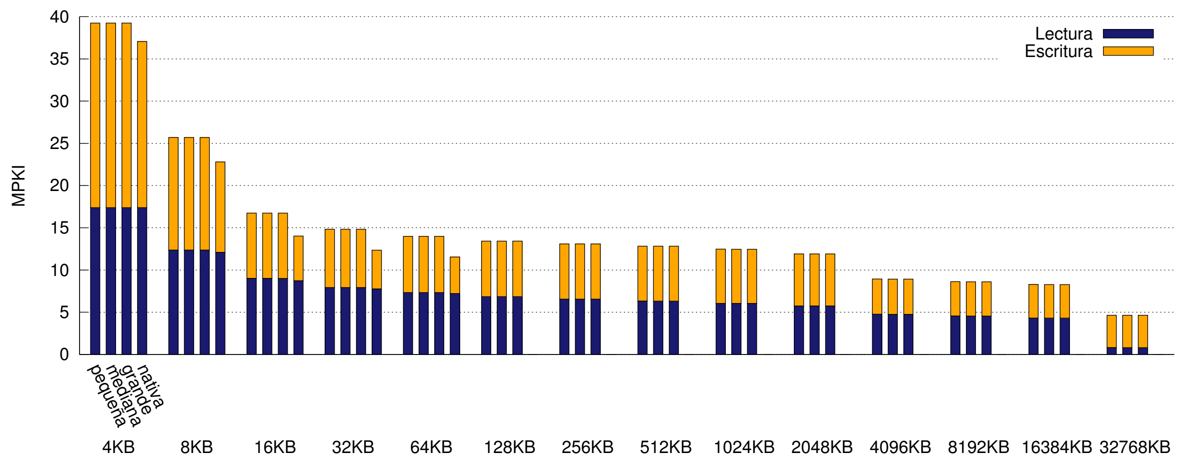


Figura D.17: Fallos por cada mil instrucciones en la cache de datos para `facesim` ejecutado en Intel. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.



(a) write-allocate y copy-back, sólo instrucciones de usuario



(b) non-write-allocate y write-through, instrucciones de usuario y sistema

Figura D.18: Fallos por cada mil instrucciones en la cache de datos para `facesim` ejecutado en Sparc.

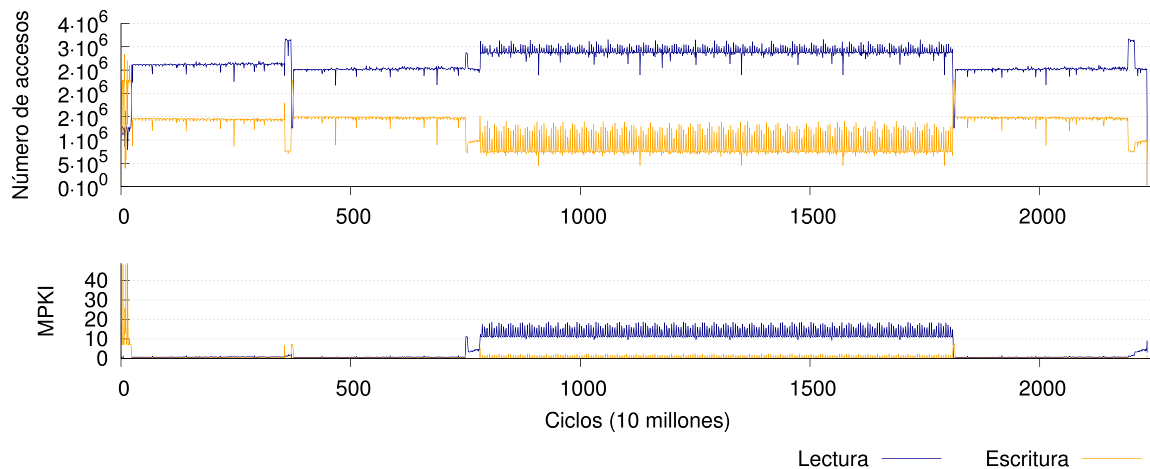


Figura D.19: Traza temporal de fallos en cache para `dedup` con entrada grande (igual a la pequeña y mediana), ejecutado en Sparc. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

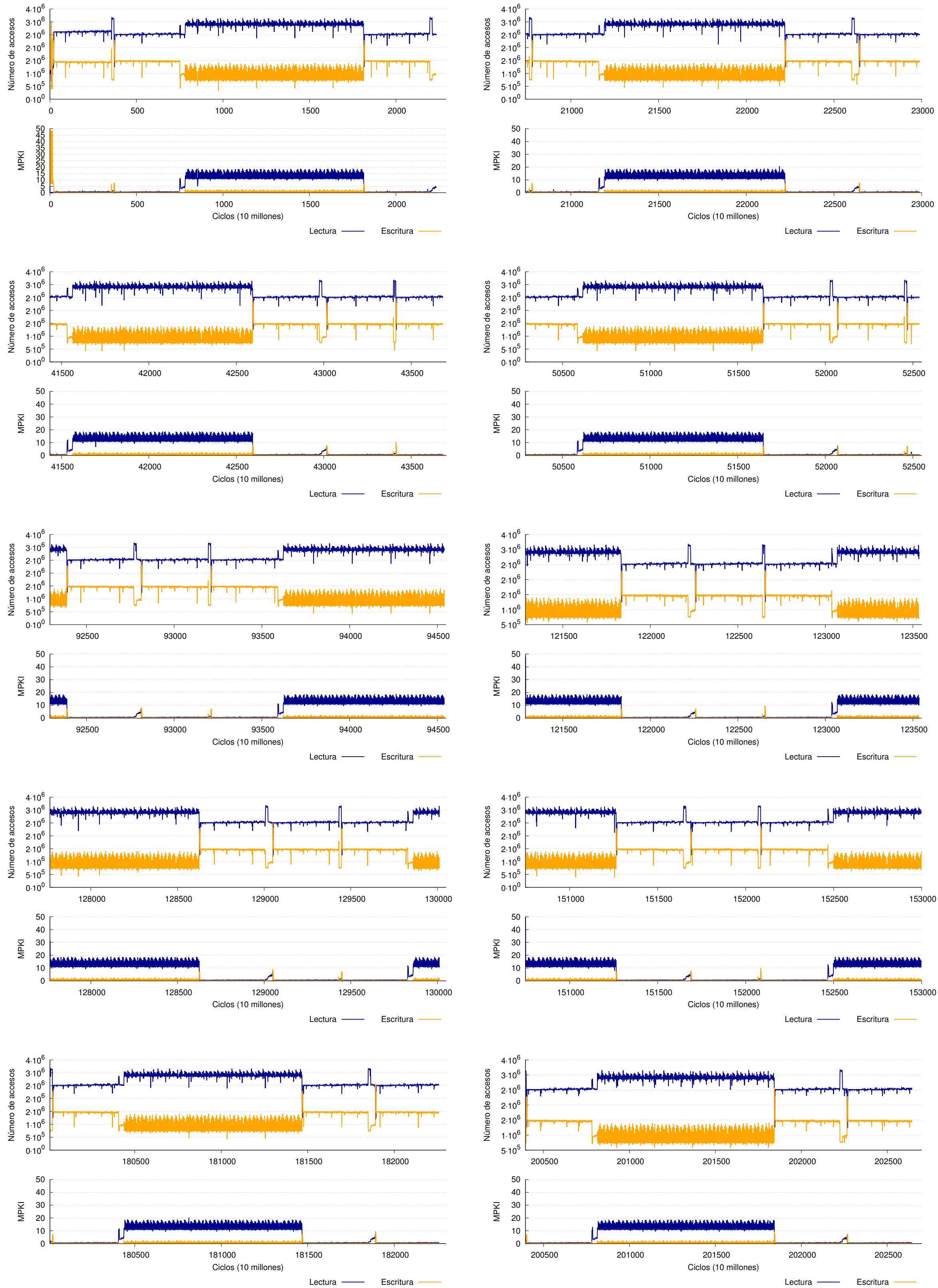


Figura D.20: Traza temporal de fallos en cache para `facesim` con entrada nativa, ejecutado en Sparc. Aparecen diez muestras tomadas al azar del total de la ejecución. Se utiliza una política write-allocate y copy-back.

Ferret

En las figuras D.21 y D.22 vemos que el número de fallos crece al pasar de la entrada pequeña a la mediana y de la mediana a la grande, pero que la entrada nativa tiene un número menor de fallos en todos los casos. Por lo tanto, para este benchmark, no se cumple que las entradas mayores estresen más la jerarquía de memoria. La entrada pequeña es una muy buena aproximación del comportamiento de la entrada nativa.

Las trazas temporales (figura D.24) no nos aportan en este caso información demasiado valiosa, ya que no se detecta ningún patrón ni se mantienen constantes en ningún valor.

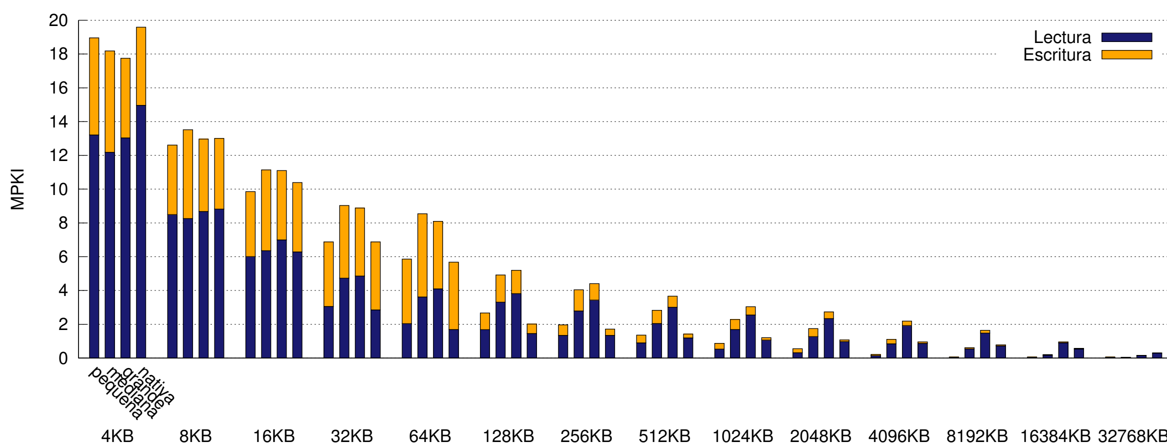
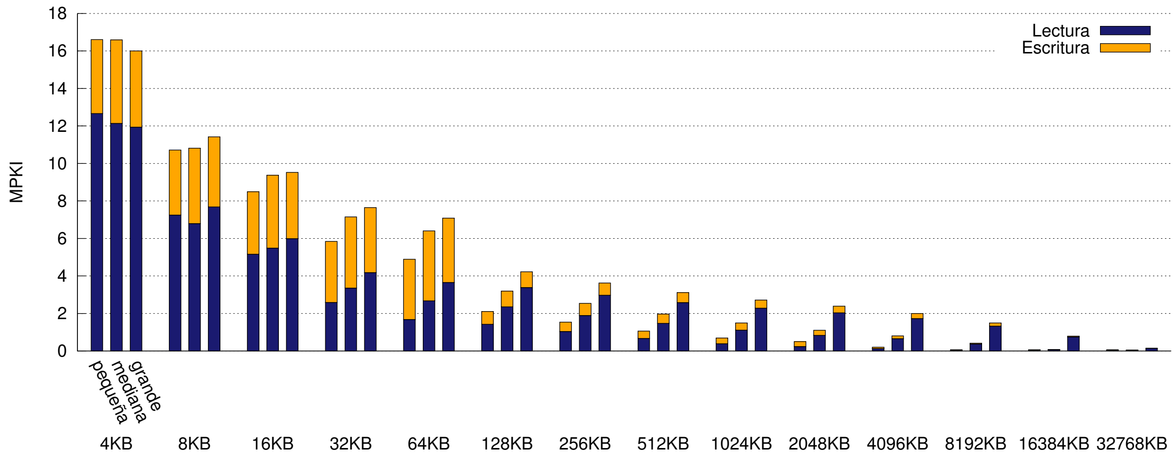


Figura D.21: Fallos por cada mil instrucciones en la cache de datos para `ferret` ejecutado en Intel. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

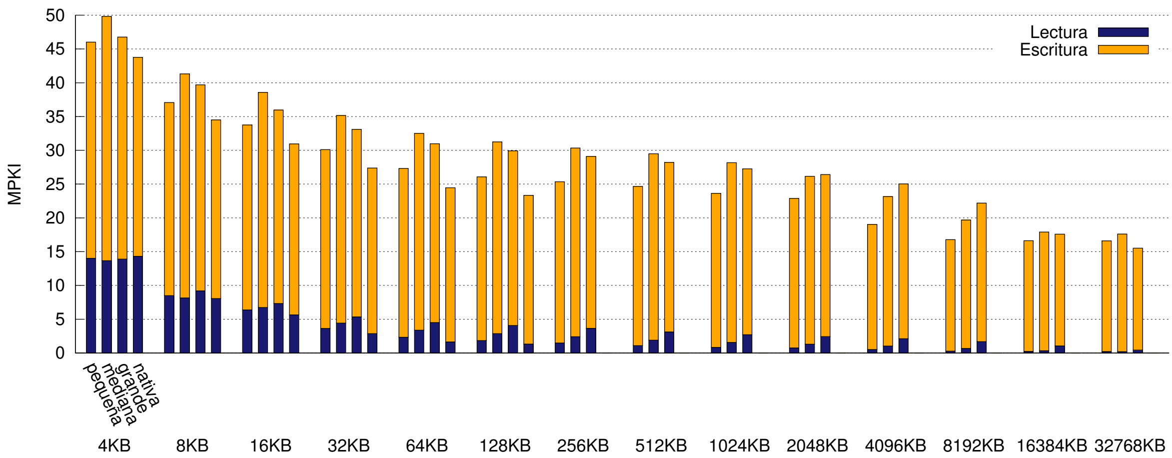
Fluidanimate

En la figura D.25 vemos que los fallos por cada mil instrucciones en Intel para la entrada nativa son muy similares a los de la entrada grande o la mediana para muchos de los tamaños de cache. De manera similar, en Sparc (figura D.26), se acercan a los que presenta la entrada de tamaño medio. En los dos casos parece que, con los tamaños de cache más grande, las entradas más pequeñas presentan menor número de fallos, similar a lo que sucedía para `blackscholes` de manera más evidente.

Para esta aplicación, la traza temporal nos aporta también información fundamental para comprender el funcionamiento de la aplicación. Con las entradas pequeña, mediana y grande (figura D.27) se aprecia claramente un patrón que se repite cinco veces, aunque al aumentar el tamaño de la entrada el mismo patrón ocupa un mayor número de ciclos. En la entrada nativa (figura D.28) vemos el mismo patrón, que ahora ocupa mayor número de ciclos y se repite más veces.

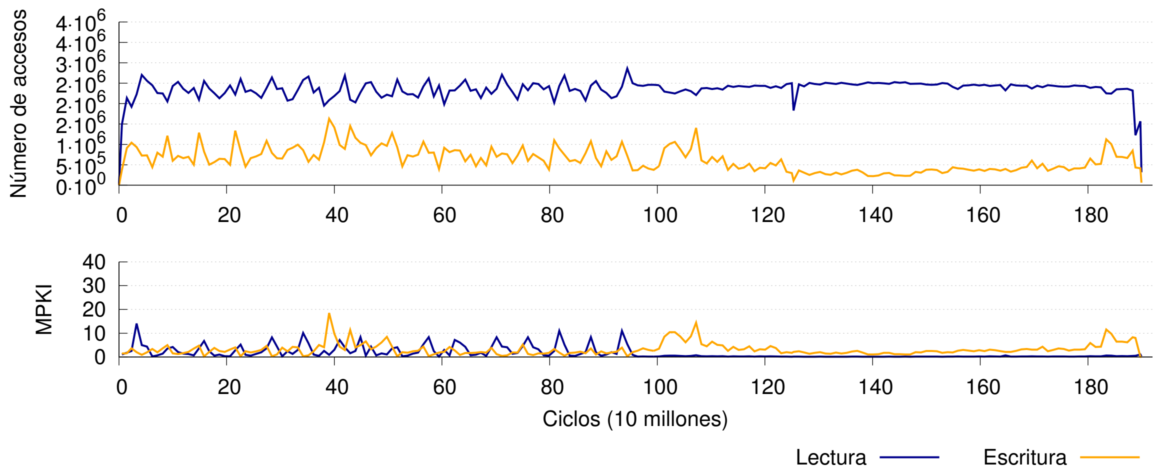


(a) write-allocate y copy-back, sólo instrucciones de usuario

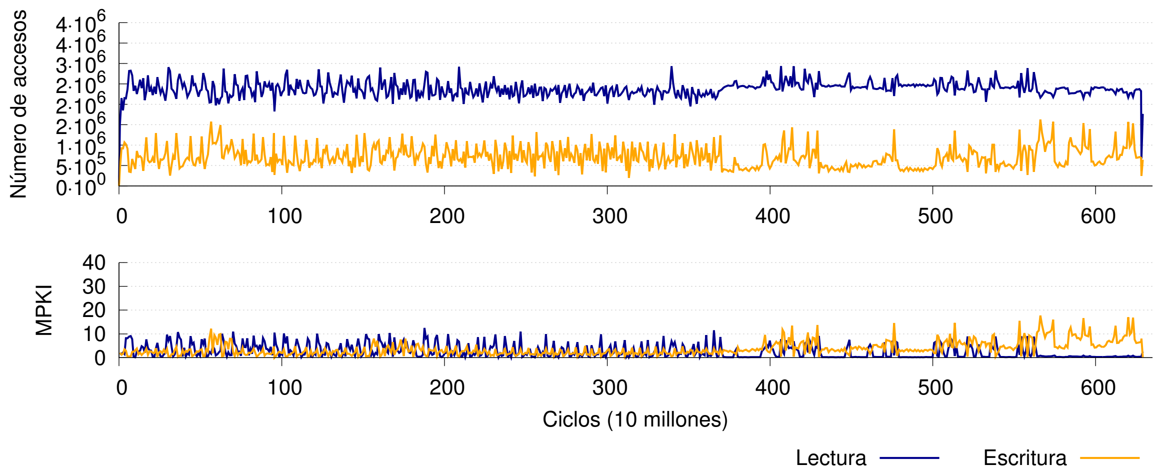


(b) non-write-allocate y write-through, instrucciones de usuario y sistema

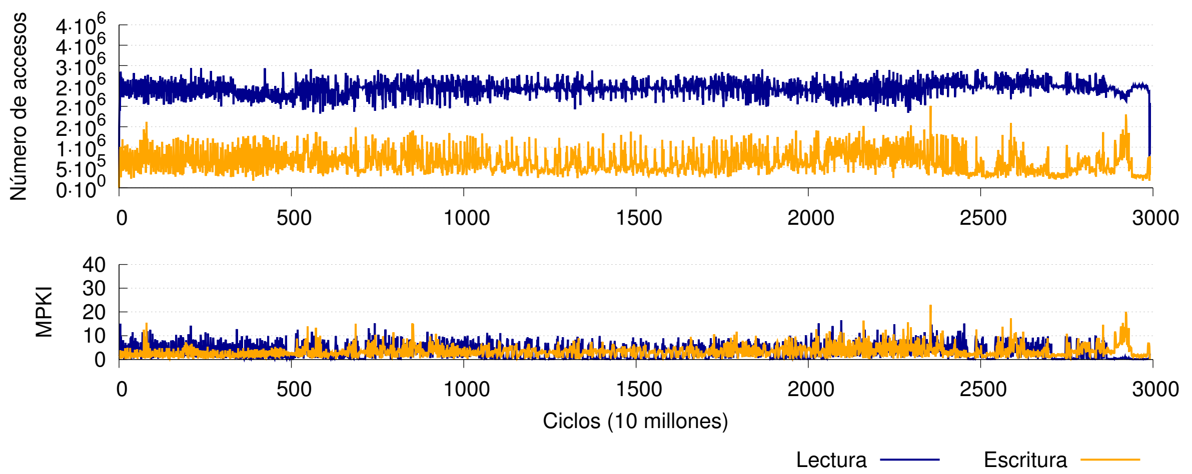
Figura D.22: Fallos por cada mil instrucciones en la cache de datos para `ferret` ejecutado en Sparc.



(a) pequeña



(b) mediana



(c) grande

Figura D.23: Traza temporal de fallos en cache para `ferret` con entradas pequeña, mediana y grande, ejecutado en Sparc. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

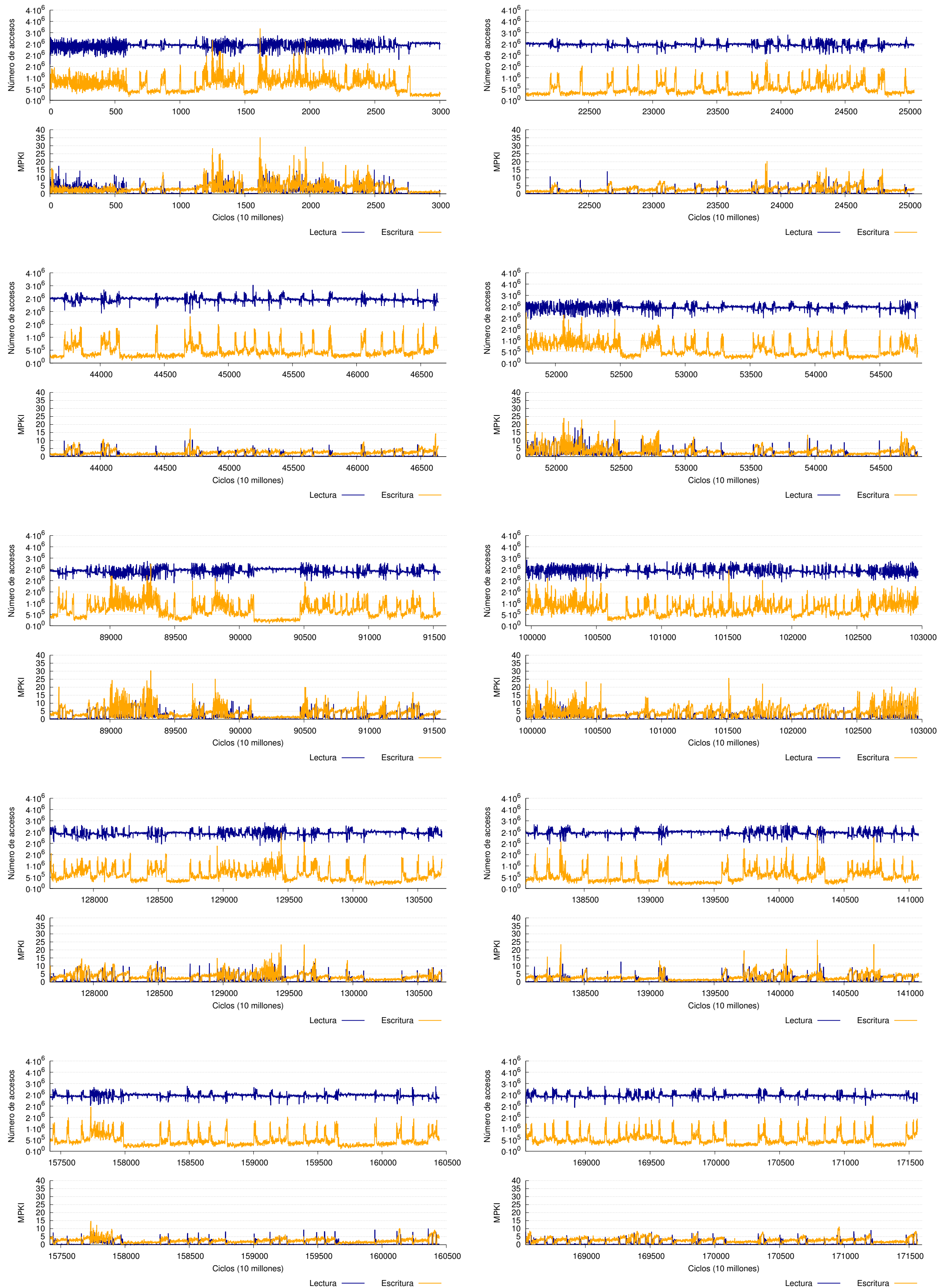


Figura D.24: Traza temporal de fallos en cache para `ferret` con entrada nativa, ejecutado en Sparc. Aparecen diez muestras tomadas al azar del total de la ejecución. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

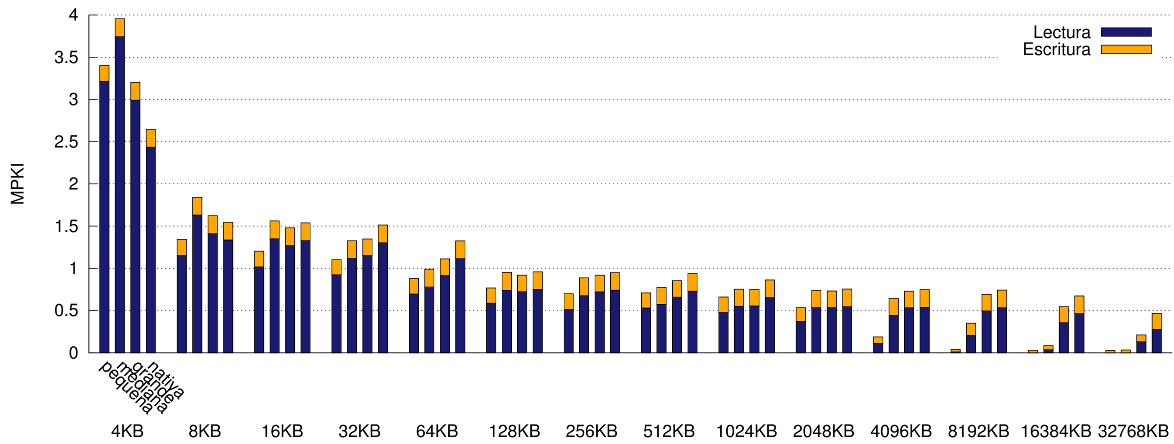
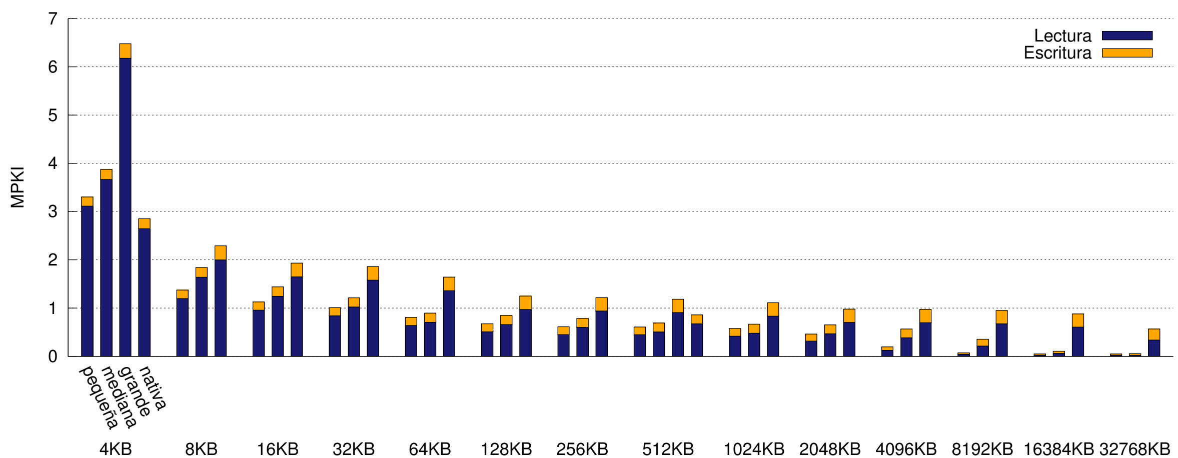
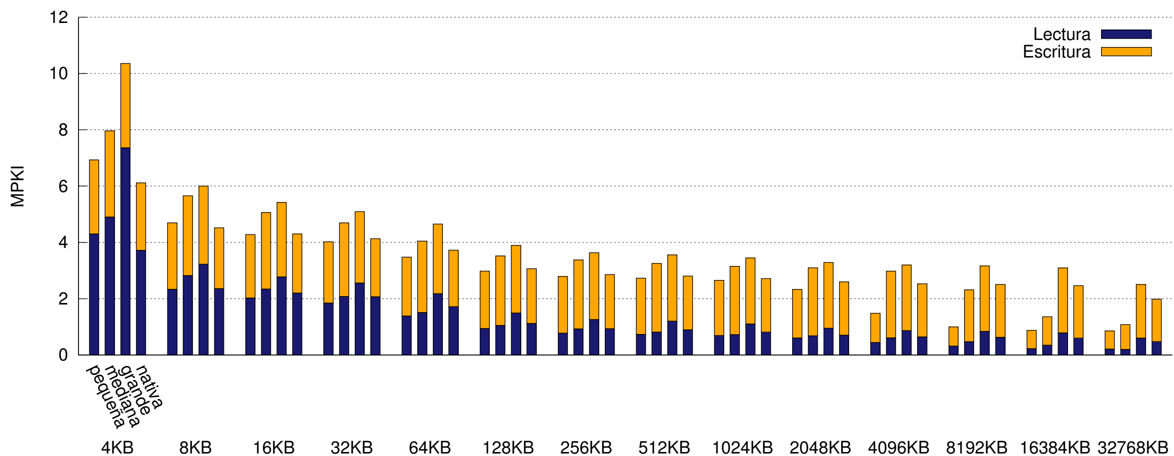


Figura D.25: Fallos por cada mil instrucciones en la cache de datos para `fluidanimate` ejecutado en Intel. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

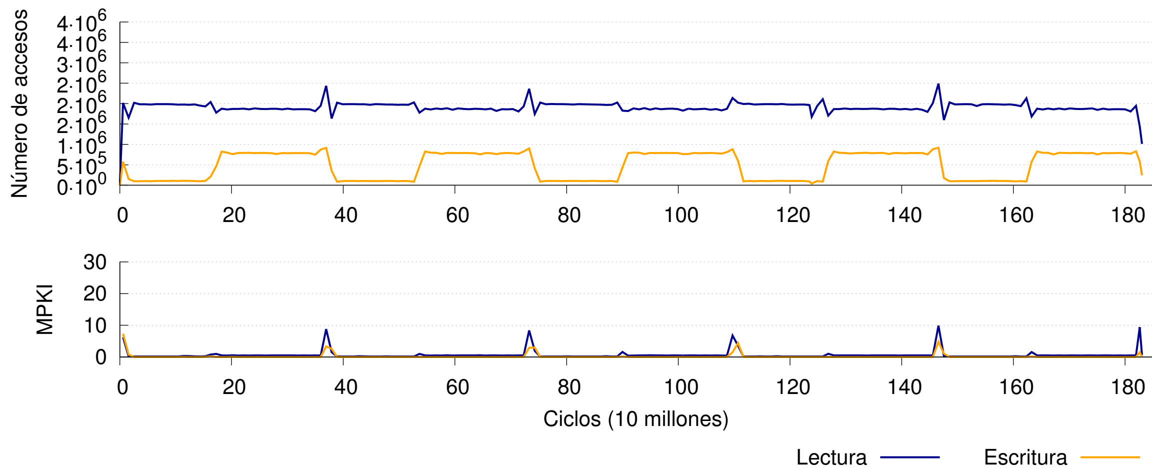


(a) write-allocate y copy-back, sólo instrucciones de usuario

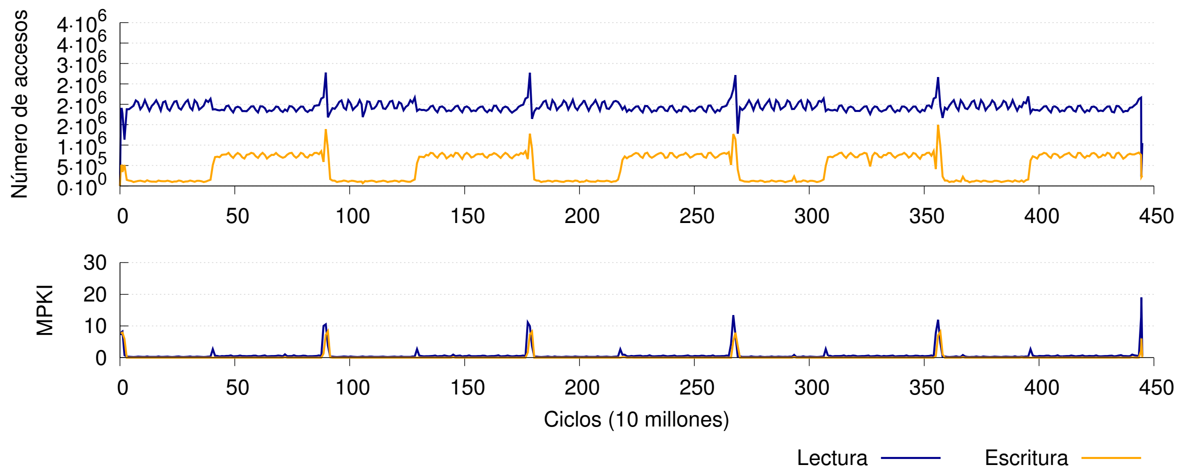


(b) non-write-allocate y write-through, instrucciones de usuario y sistema

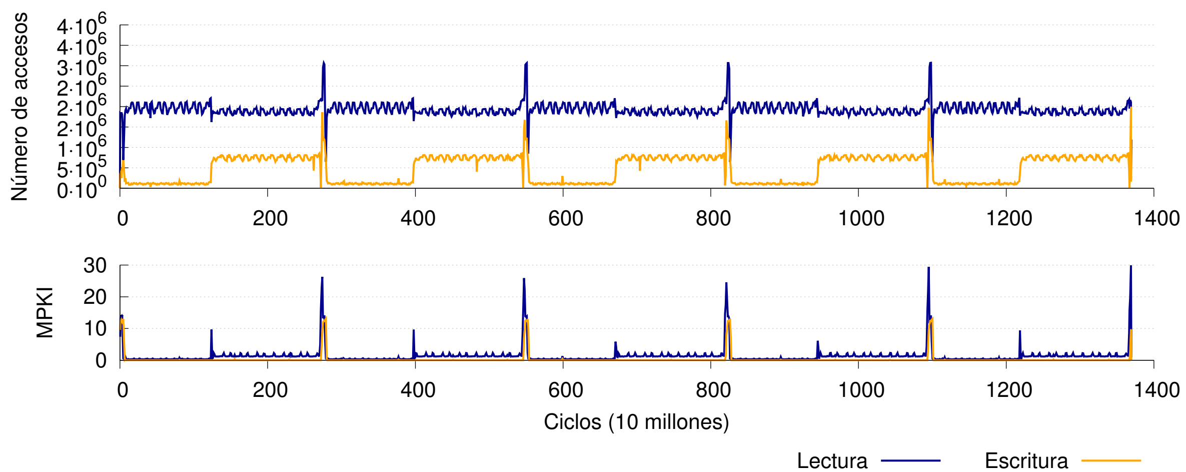
Figura D.26: Fallos por cada mil instrucciones en la cache de datos para `fluidanimate` ejecutado en Sparc.



(a) pequeña



(b) mediana



(c) grande

Figura D.27: Traza temporal de fallos en cache para `fluidanimate` con entradas pequeña, mediana y grande, ejecutado en Sparc. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

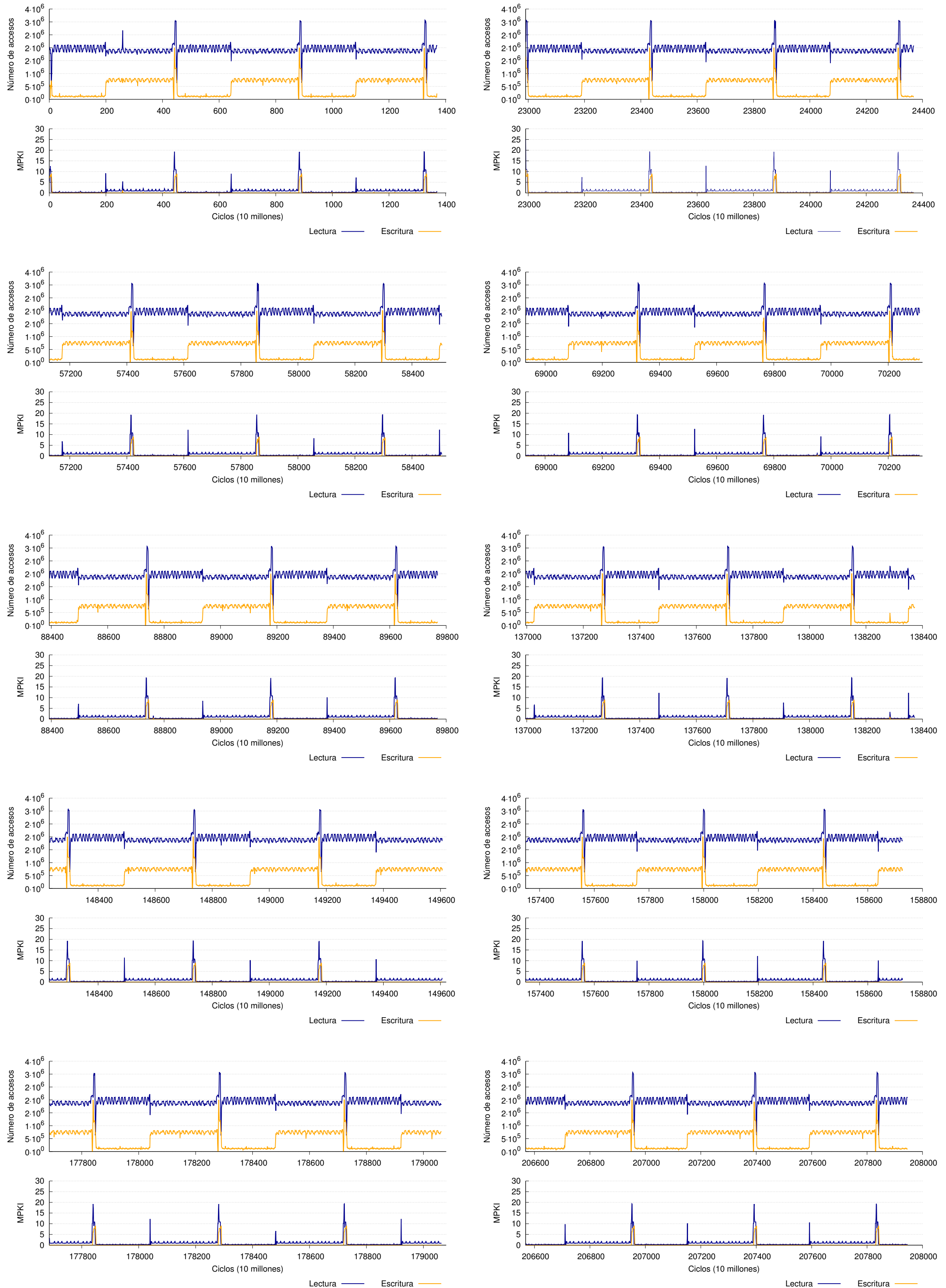


Figura D.28: Traza temporal de fallos en cache para `fluidanimate` con entrada nativa, ejecutado en Sparc. Aparecen diez muestras tomadas al azar del total de la ejecución. Se utiliza una política write-allocate y copy-back.

Freqmine

En `freqmine` nos encontramos con un caso en el que el número de fallos por cada mil instrucciones es menor según aumentamos el tamaño de la entrada (figuras D.29, D.30 y D.31). Esto va totalmente en contra de las suposiciones que se hacen siempre respecto al escalado de las entradas.

En la traza temporal (figuras D.31 y D.32) sí que se aprecian zonas diferenciadas en las que se mantiene un número mayor o menor de fallos, pero no ha sido posible encontrar un patrón claramente definido ni una relación con la entrada.

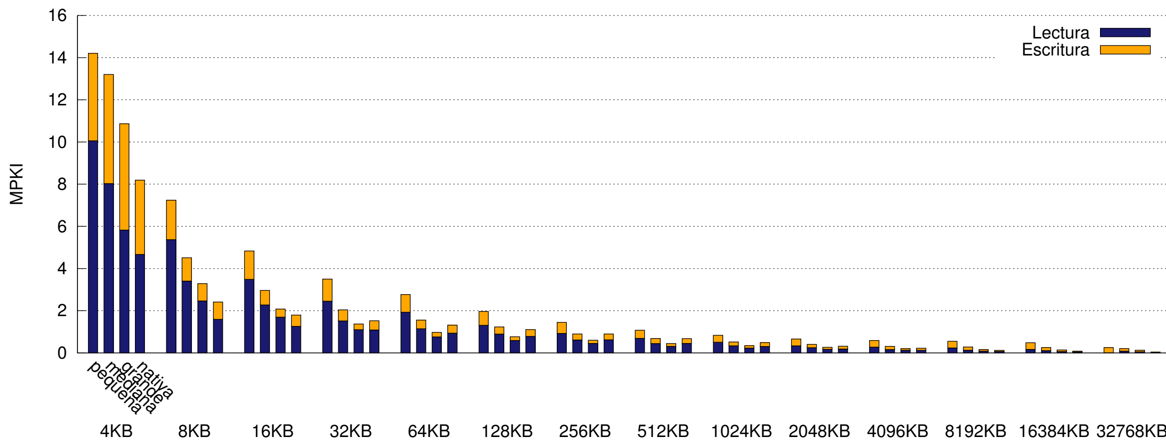
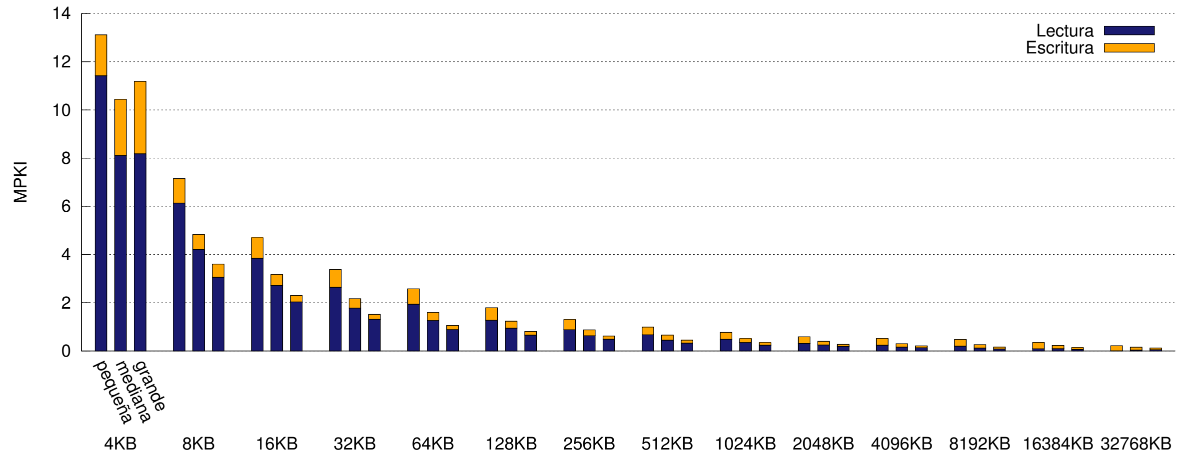


Figura D.29: Fallos por cada mil instrucciones en la cache de datos para `freqmine` ejecutado en Intel. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

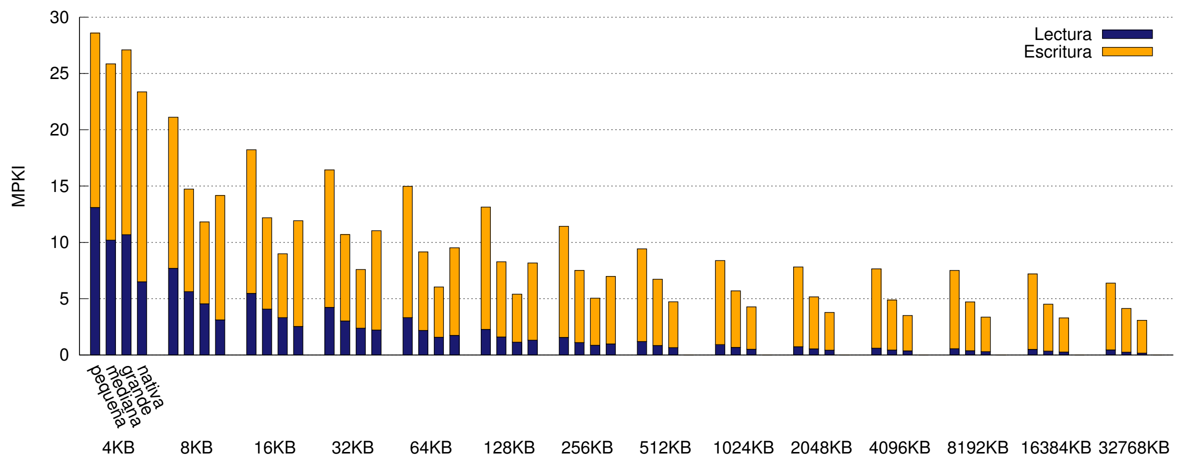
Raytrace

Lo primero que podemos detectar analizando las figuras D.33 y D.34 es que con caches mayores de 16 KB, un tamaño muy pequeño, la aplicación no presenta apenas ningún fallo (a excepción de los fallos de escritura al utilizar una política non-write-allocate, que recordamos que se deben a que los bloques nunca se traen a memoria y por lo tanto no se explota la localidad espacial ni temporal). En consecuencia, esta aplicación no será adecuada en absoluto para realizar un estudio de la jerarquía de memoria.

De todas formas, también obtenemos información interesante estudiando la traza temporal. En las entradas pequeña, mediana y grande (figura D.35) no se aprecia con claridad, pero en la entrada nativa (figura D.36) podemos ver un patrón, especialmente fijándonos en el número de instrucciones de lectura ejecutadas.

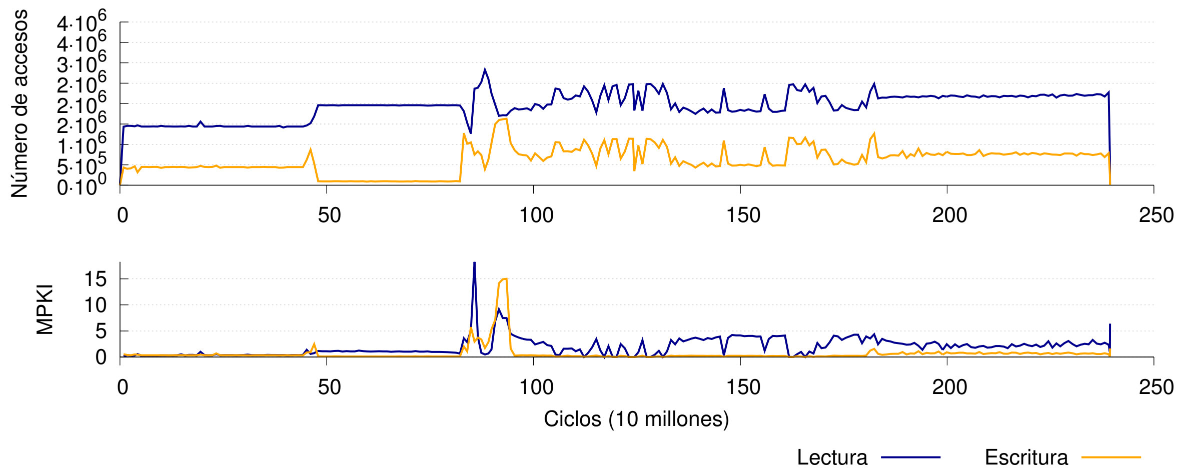


(a) write-allocate y copy-back, sólo instrucciones de usuario

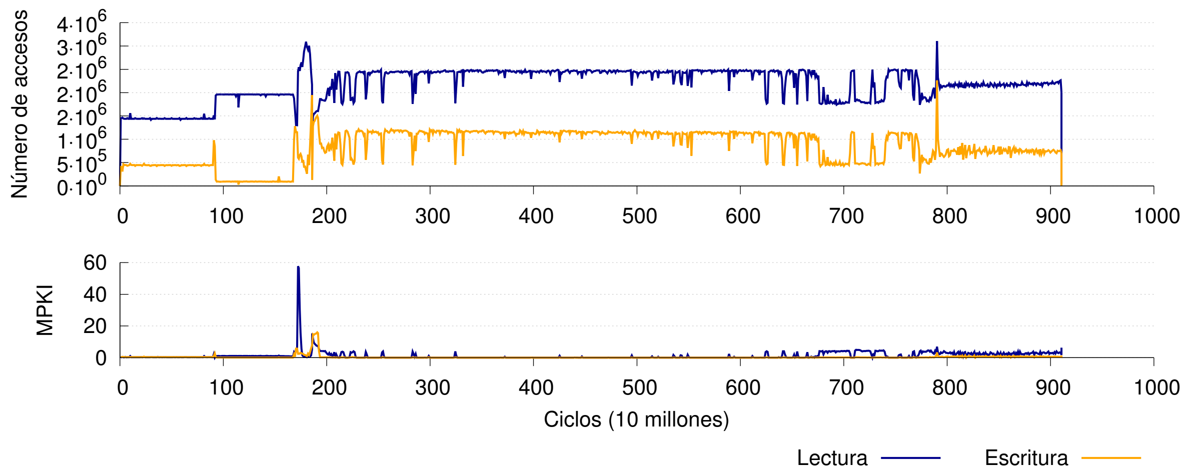


(b) non-write-allocate y write-through, instrucciones de usuario y sistema

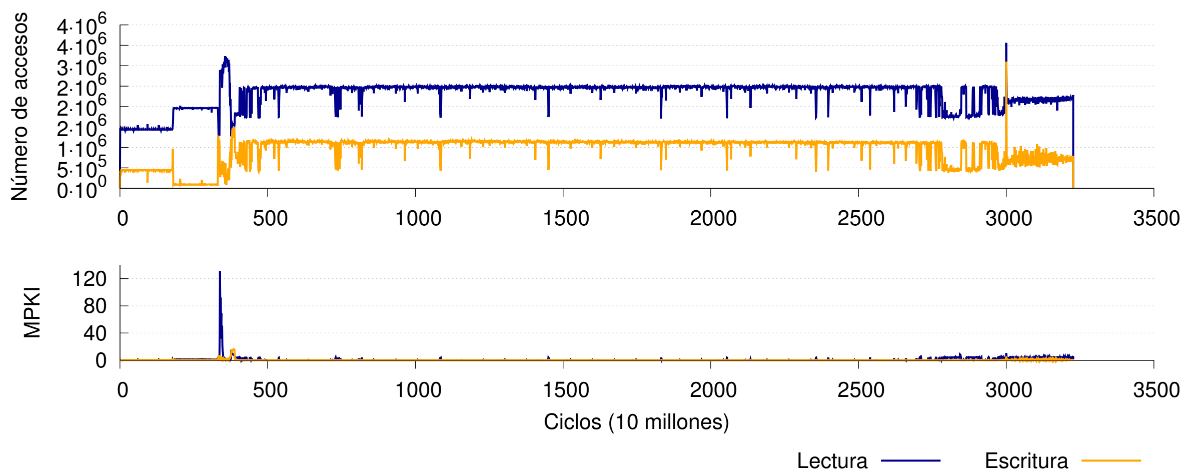
Figura D.30: Fallos por cada mil instrucciones en la cache de datos para `freqmine` ejecutado en Sparc.



(a) pequeña



(b) mediana



(c) grande

Figura D.31: Traza temporal de fallos en cache para `freqmine` con entradas pequeña, mediana y grande, ejecutado en Sparc. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

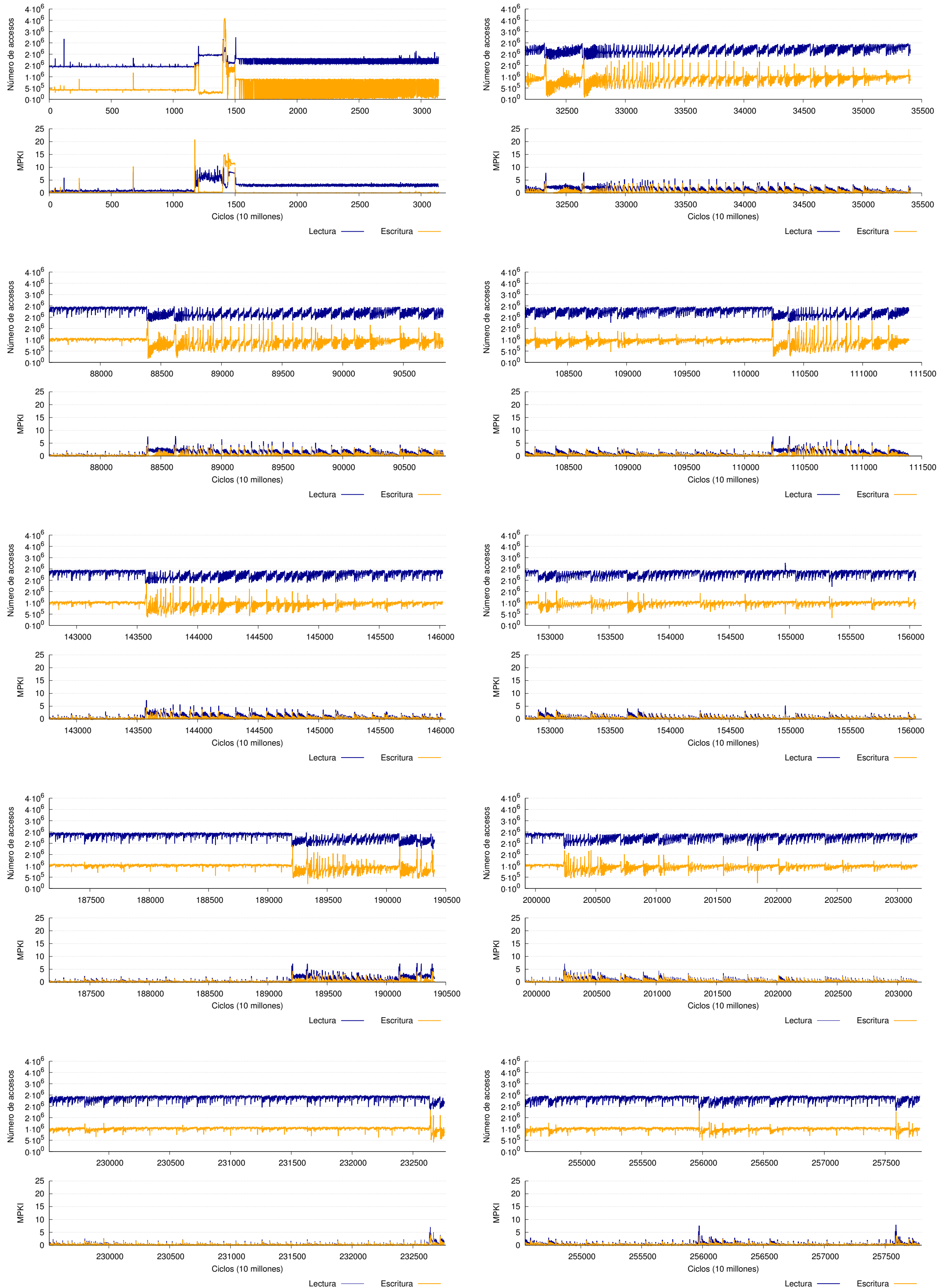


Figura D.32: Traza temporal de fallos en cache para `freqmine` con entrada nativa, ejecutado en Sparc. Aparecen diez muestras tomadas al azar del total de la ejecución. Se utiliza una política write-allocate y copy-back.

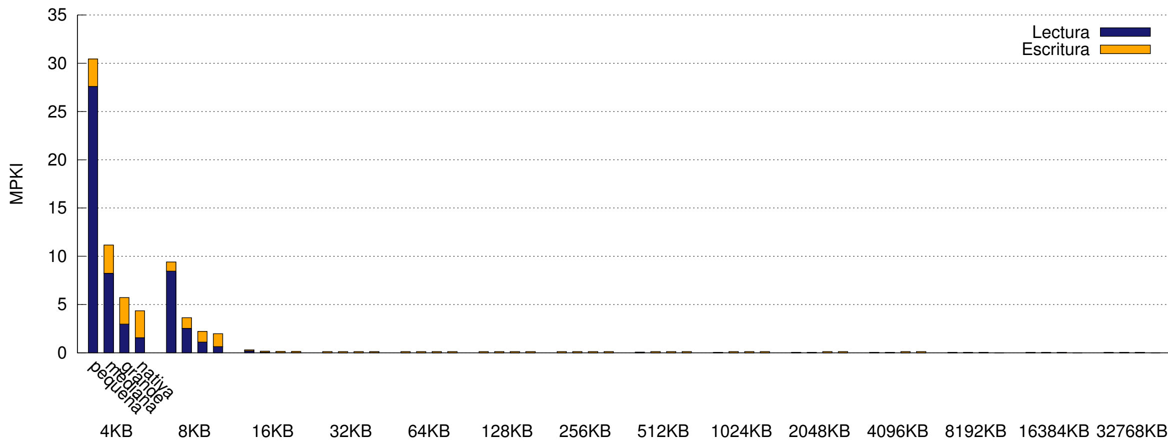
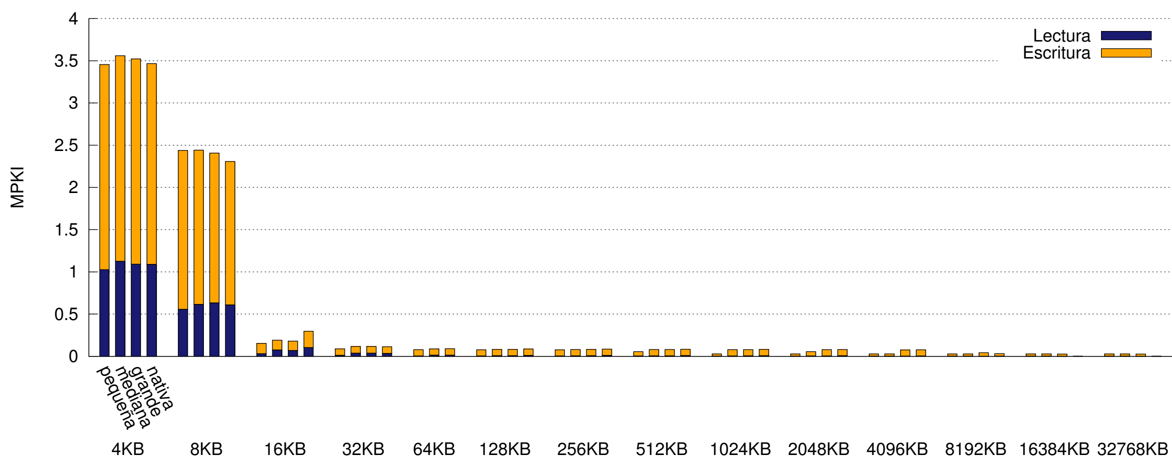
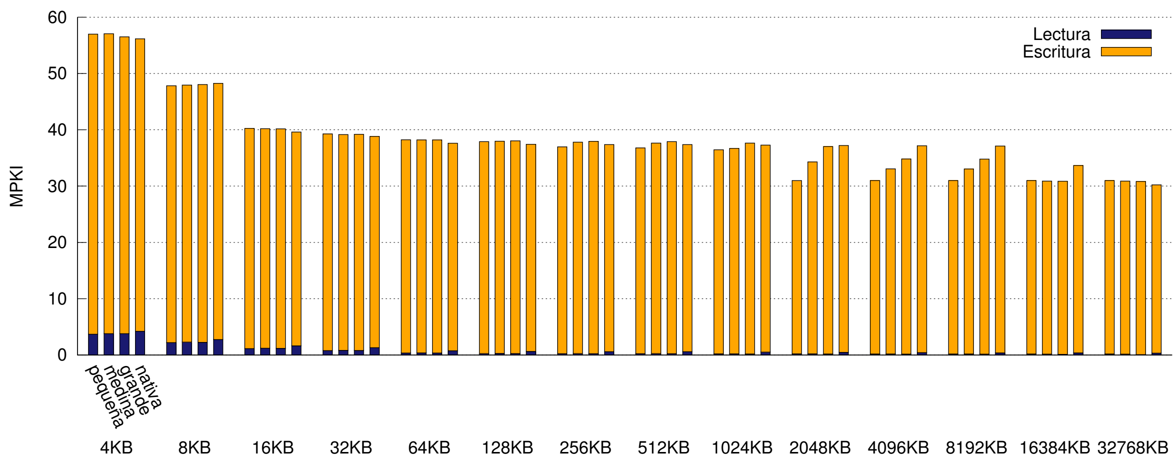


Figura D.33: Fallos por cada mil instrucciones en la cache de datos para `raytrace` ejecutado en Intel. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

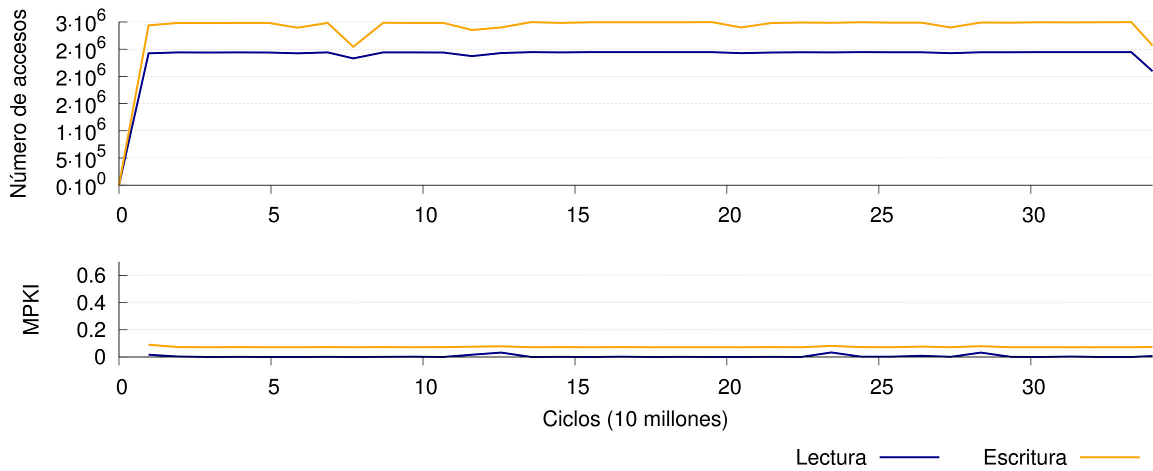


(a) write-allocate y copy-back, sólo instrucciones de usuario

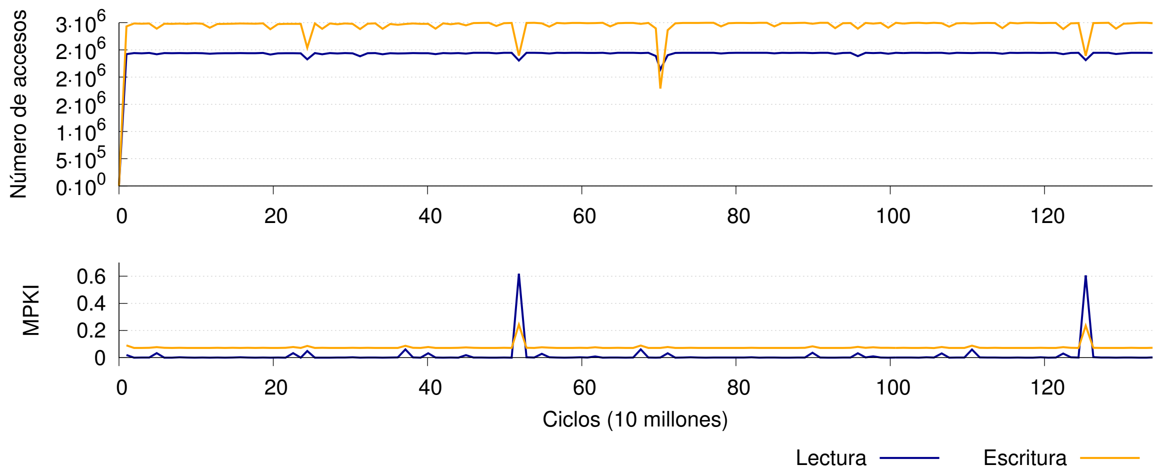


(b) non-write-allocate y write-through, instrucciones de usuario y sistema

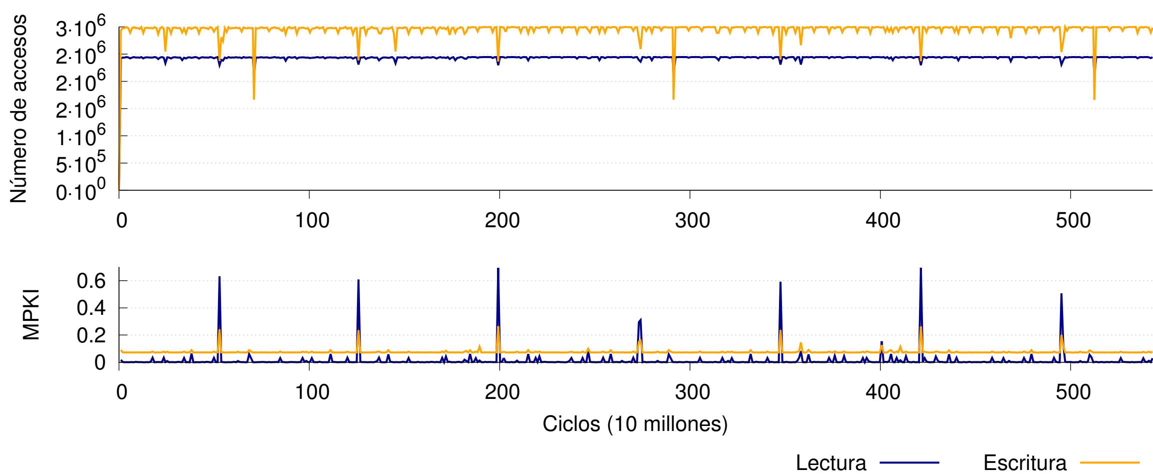
Figura D.34: Fallos por cada mil instrucciones en la cache de datos para `raytrace` ejecutado en Sparc.



(a) pequeña



(b) mediana



(c) grande

Figura D.35: Traza temporal de fallos en cache para `raytrace` con entradas pequeña, mediana y grande, ejecutado en Sparc. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

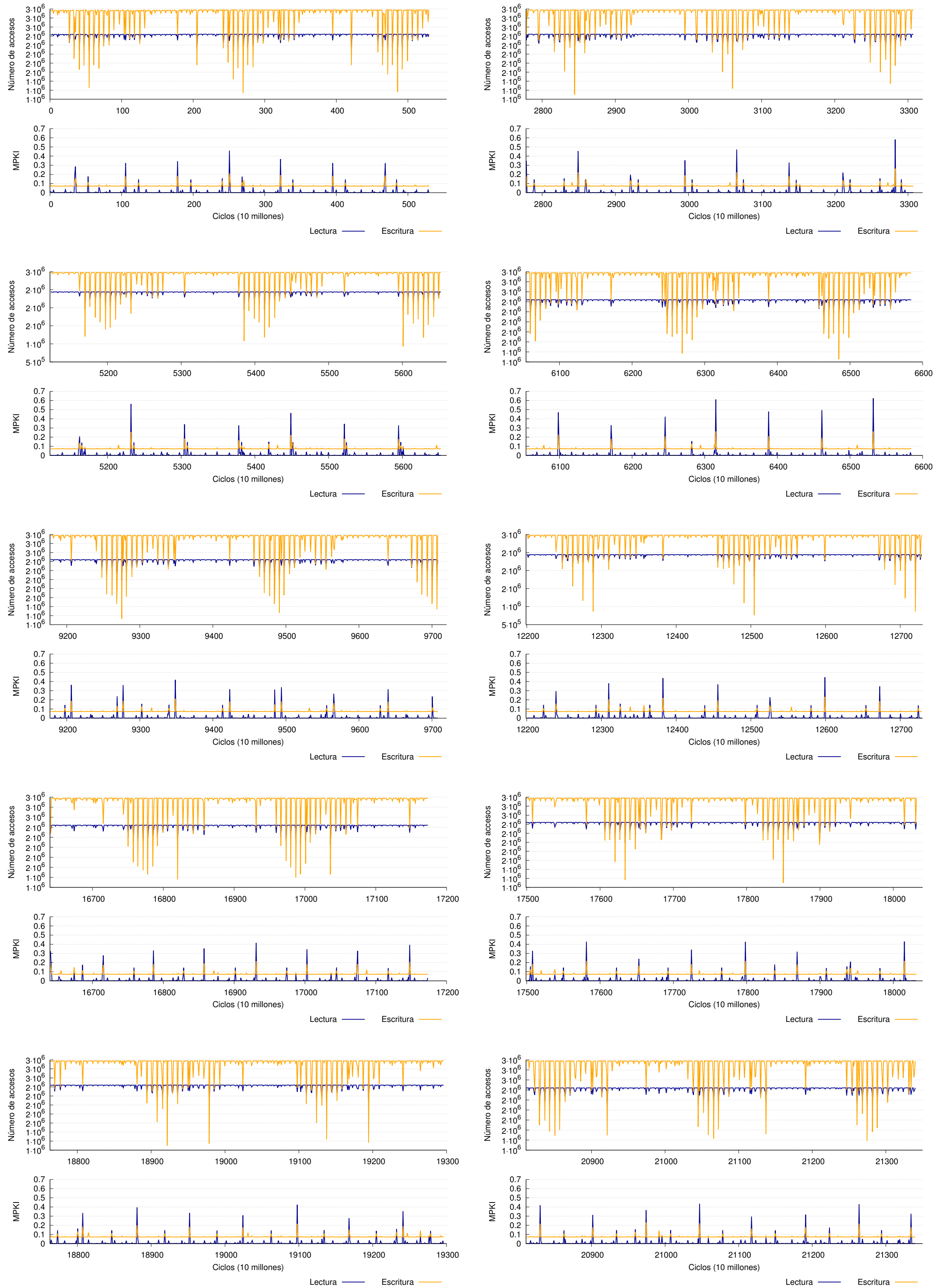


Figura D.36: Traza temporal de fallos en cache para `raytrace` con entrada nativa, ejecutado en Sparc. Aparecen diez muestras tomadas al azar del total de la ejecución. Se utiliza una política write-allocate y copy-back.

Streamcluster

En los fallos en cache de este benchmark (figuras D.37 y D.38) vemos repetido el comportamiento de `blackscholes`, explicado en la sección D.1.4. Según va aumentando la capacidad de la cache, las estructuras de datos de las entradas de menor tamaño pueden almacenarse en la cache y el número de fallos disminuye drásticamente.

Analizando las trazas temporales (figuras D.39 y D.40) vemos que, tras una pequeña zona inicial, la forma de las gráficas se mantiene constante durante el resto de la ejecución.

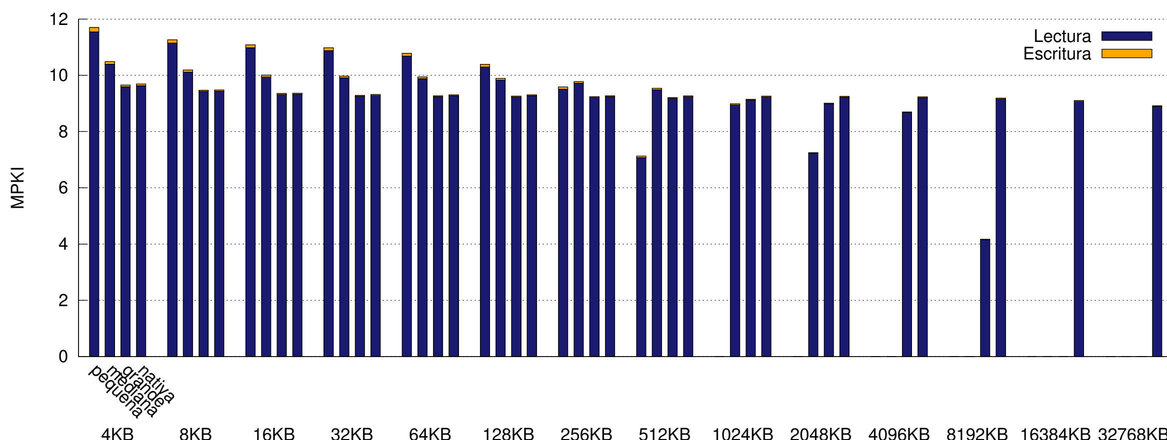
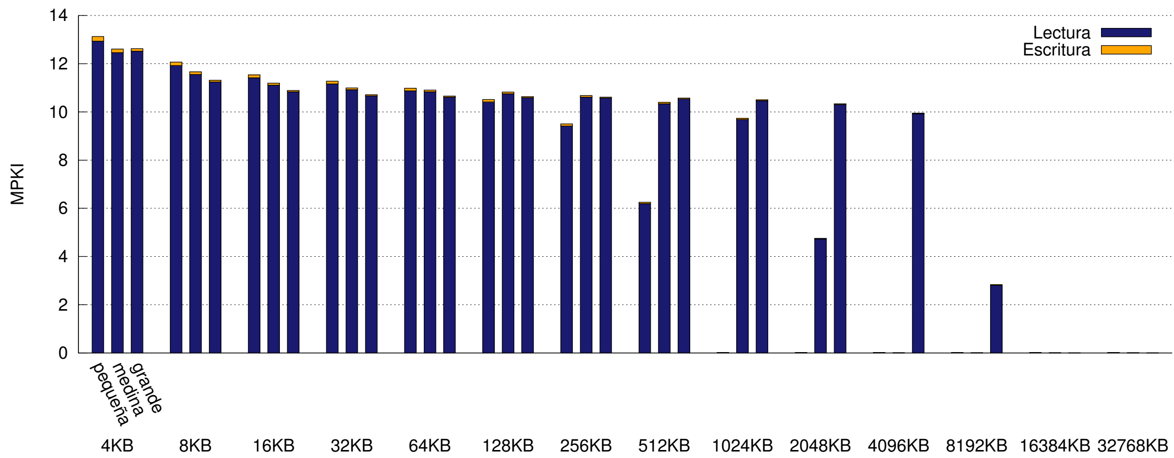


Figura D.37: Fallos por cada mil instrucciones en la cache de datos para `raytrace` ejecutado en Intel. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

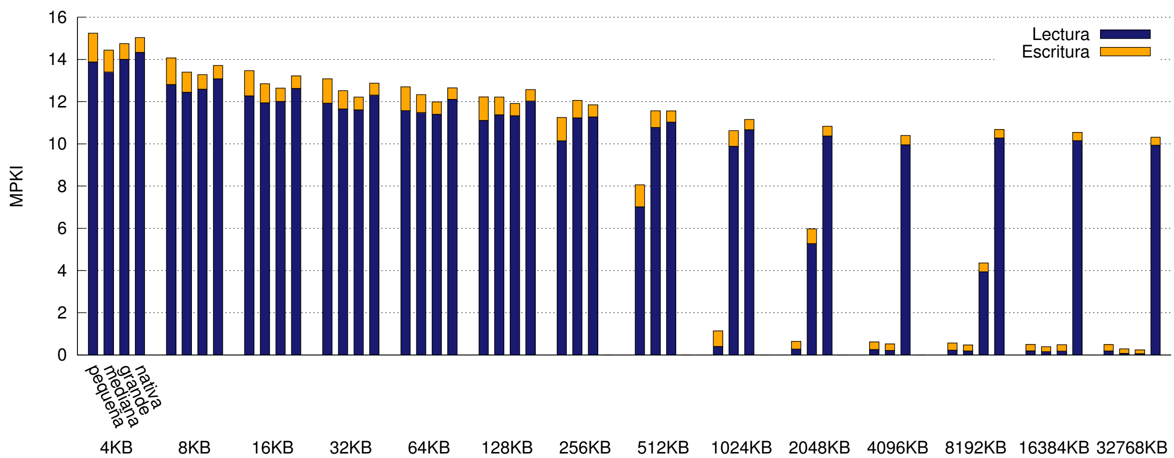
Swaptions

En las figuras D.41 y D.42 vemos rápidamente que no hay ninguna diferencia en los fallos por cada mil instrucciones de las cuatro entradas. Además, se trata de una aplicación que presenta muy pocos fallos en cache y, por lo tanto, no será útil para el estudio de la jerarquía de memoria.

Fijándonos en la traza temporal (figuras D.43 y D.44) vemos que la única diferencia entre las entradas es el número de ciclos que tardaron en ejecutarse. La tasa de fallos se mantiene constante a lo largo de la ejecución, con pequeños picos cada 750 millones de instrucciones.

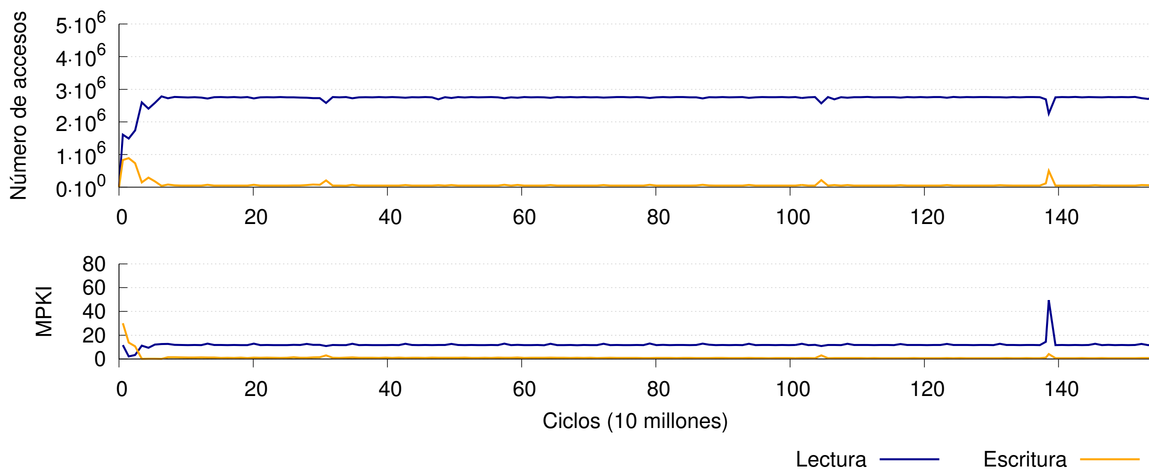


(a) write-allocate y copy-back, sólo instrucciones de usuario

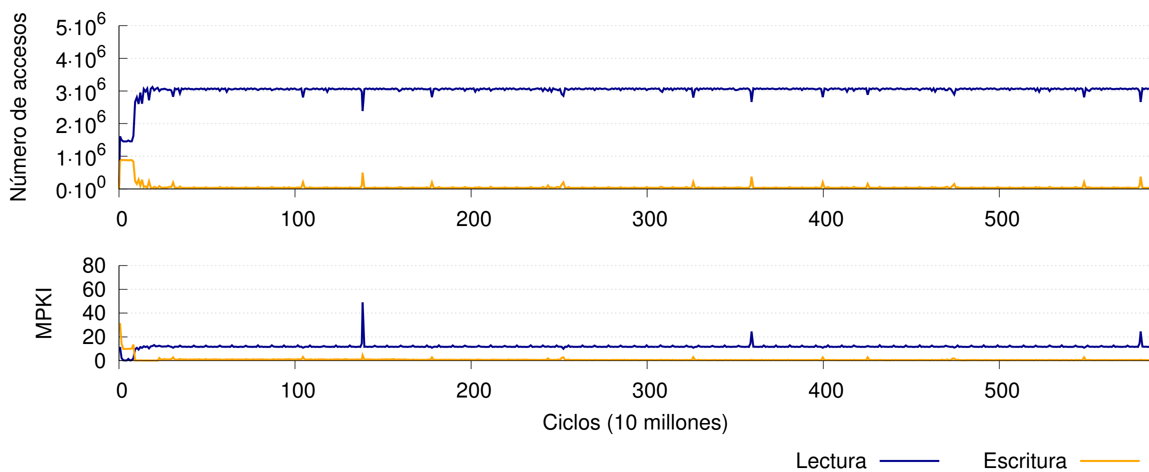


(b) non-write-allocate y write-through, instrucciones de usuario y sistema

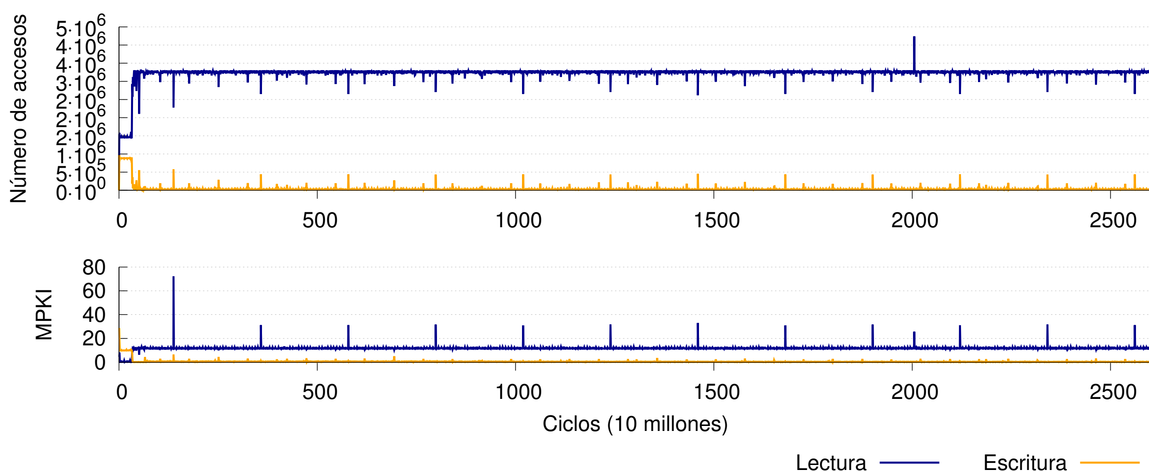
Figura D.38: Fallos por cada mil instrucciones en la cache de datos para `streamcluster` ejecutado en Sparc.



(a) pequeña



(b) mediana



(c) grande

Figura D.39: Traza temporal de fallos en cache para `streamcluster` con entradas pequeña, mediana y grande, ejecutado en Sparc. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

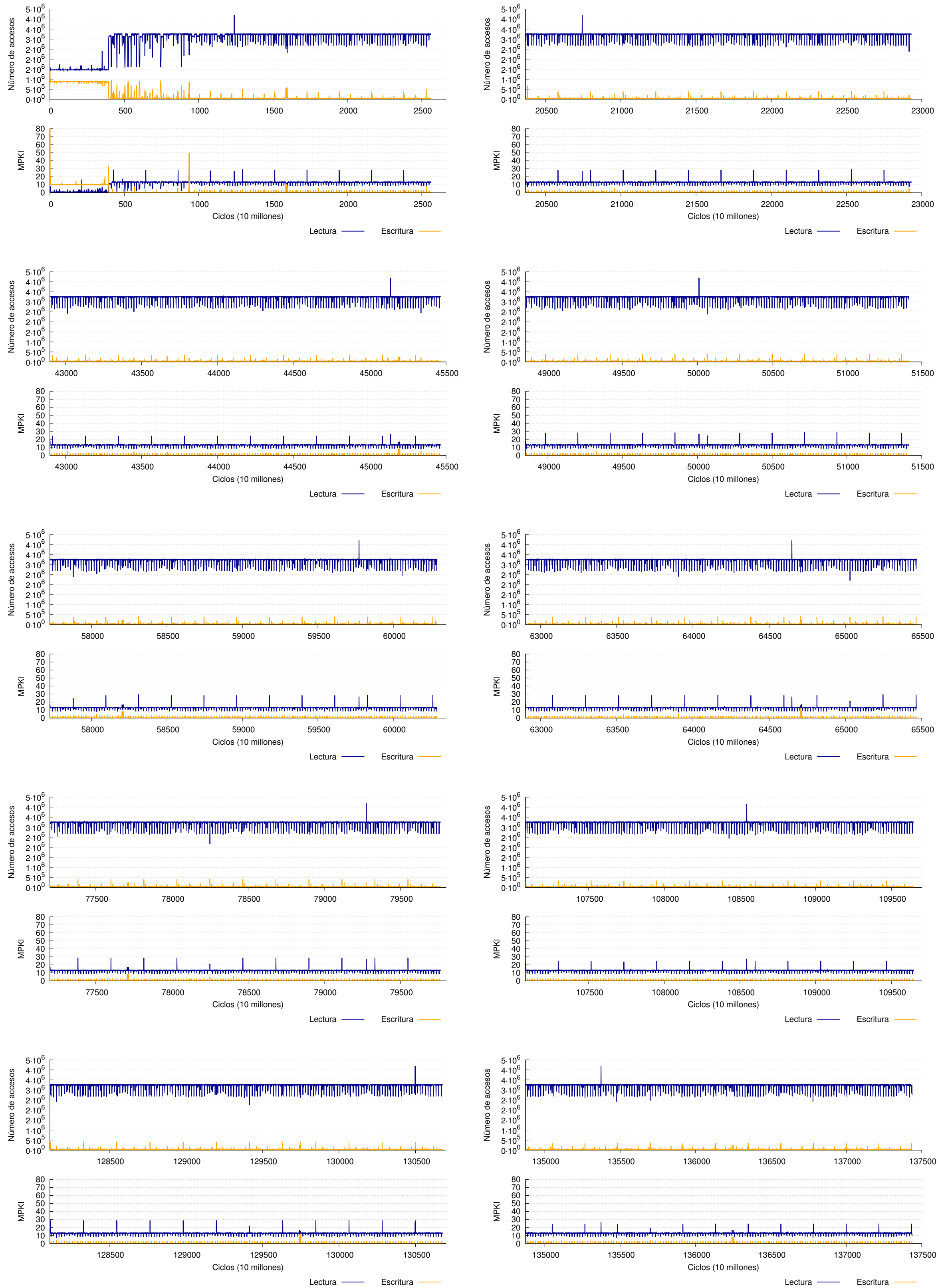


Figura D.40: Traza temporal de fallos en cache para `streamcluster` con entrada nativa, ejecutado en Sparc. Aparecen diez muestras tomadas al azar del total de la ejecución. Se utiliza una política write-allocate y copy-back.

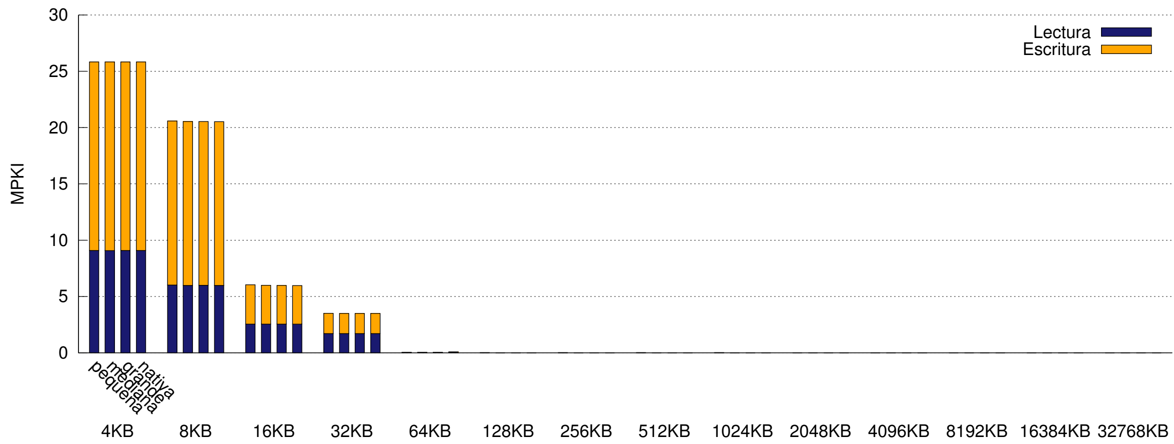
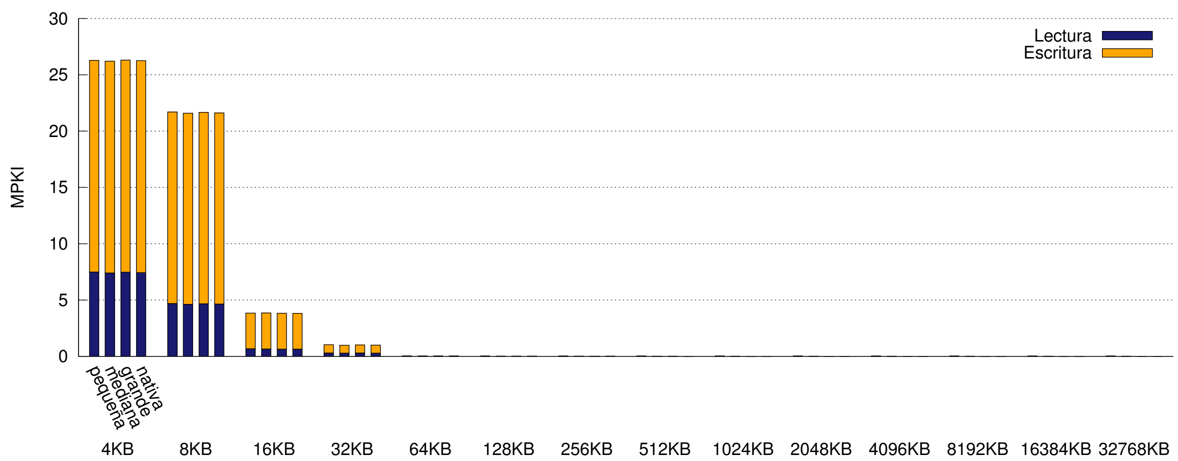
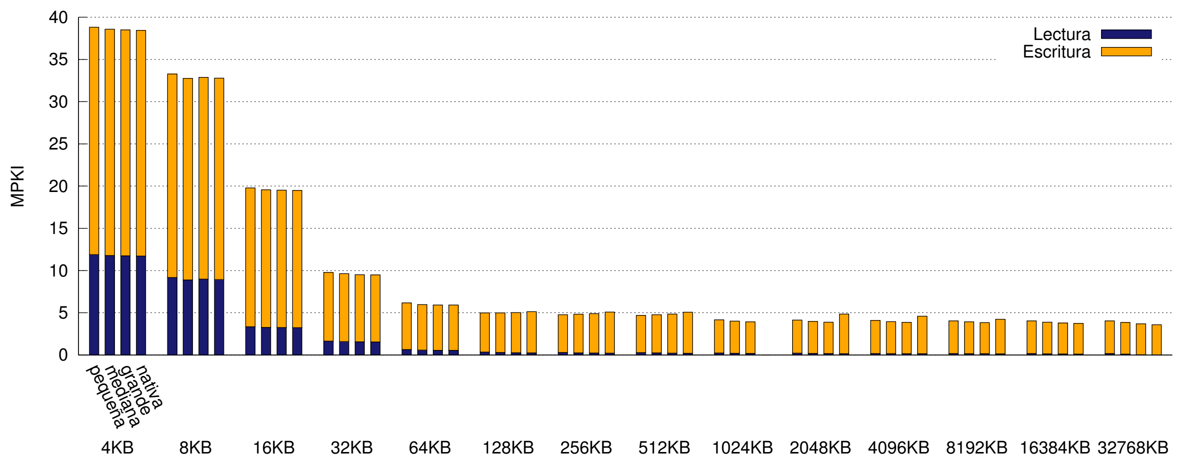


Figura D.41: Fallos por cada mil instrucciones en la cache de datos para `swaptions` ejecutado en Intel. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

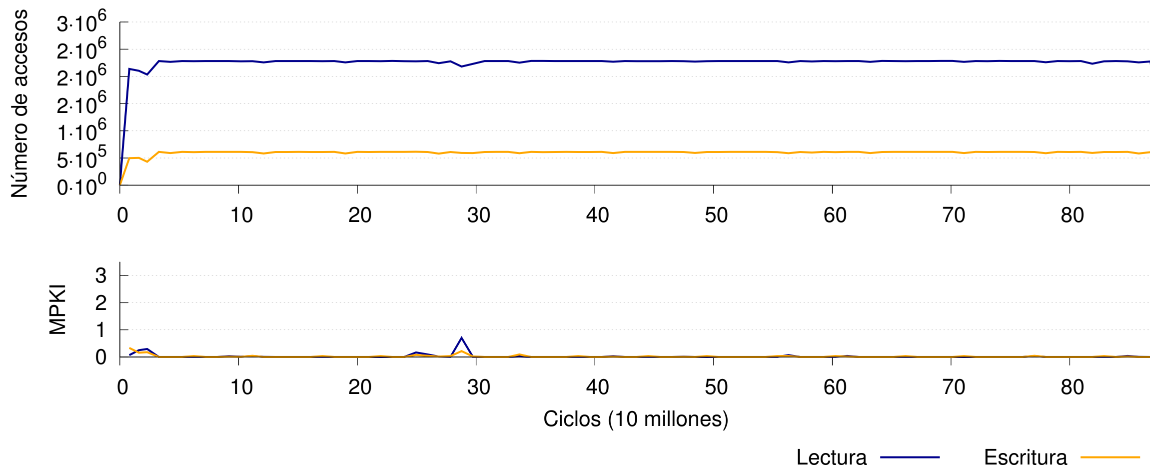


(a) write-allocate y copy-back, sólo instrucciones de usuario

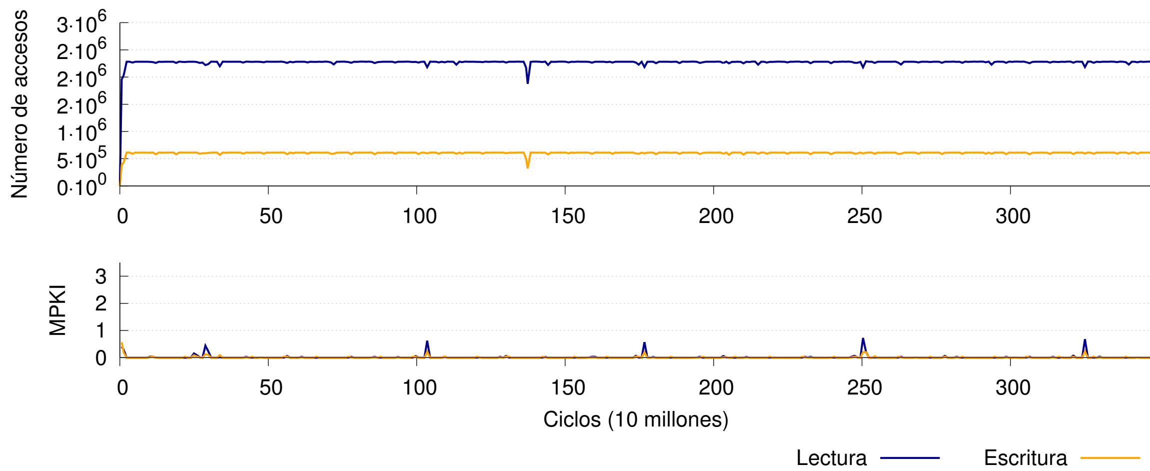


(b) non-write-allocate y write-through, instrucciones de usuario y sistema

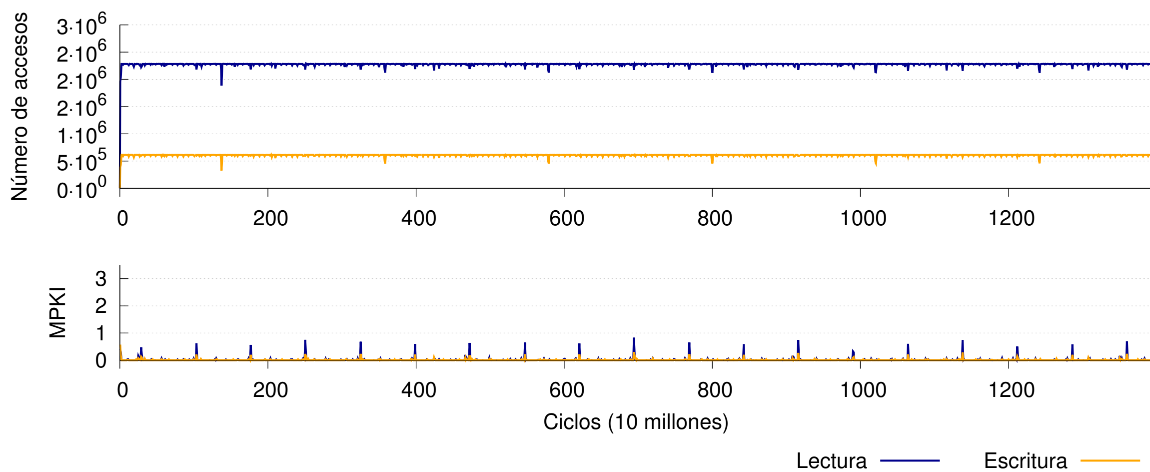
Figura D.42: Fallos por cada mil instrucciones en la cache de datos para `swaptions` ejecutado en Sparc.



(a) pequeña



(b) mediana



(c) grande

Figura D.43: Traza temporal de fallos en cache para `swaptions` con entradas pequeña, mediana y grande, ejecutado en Sparc. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

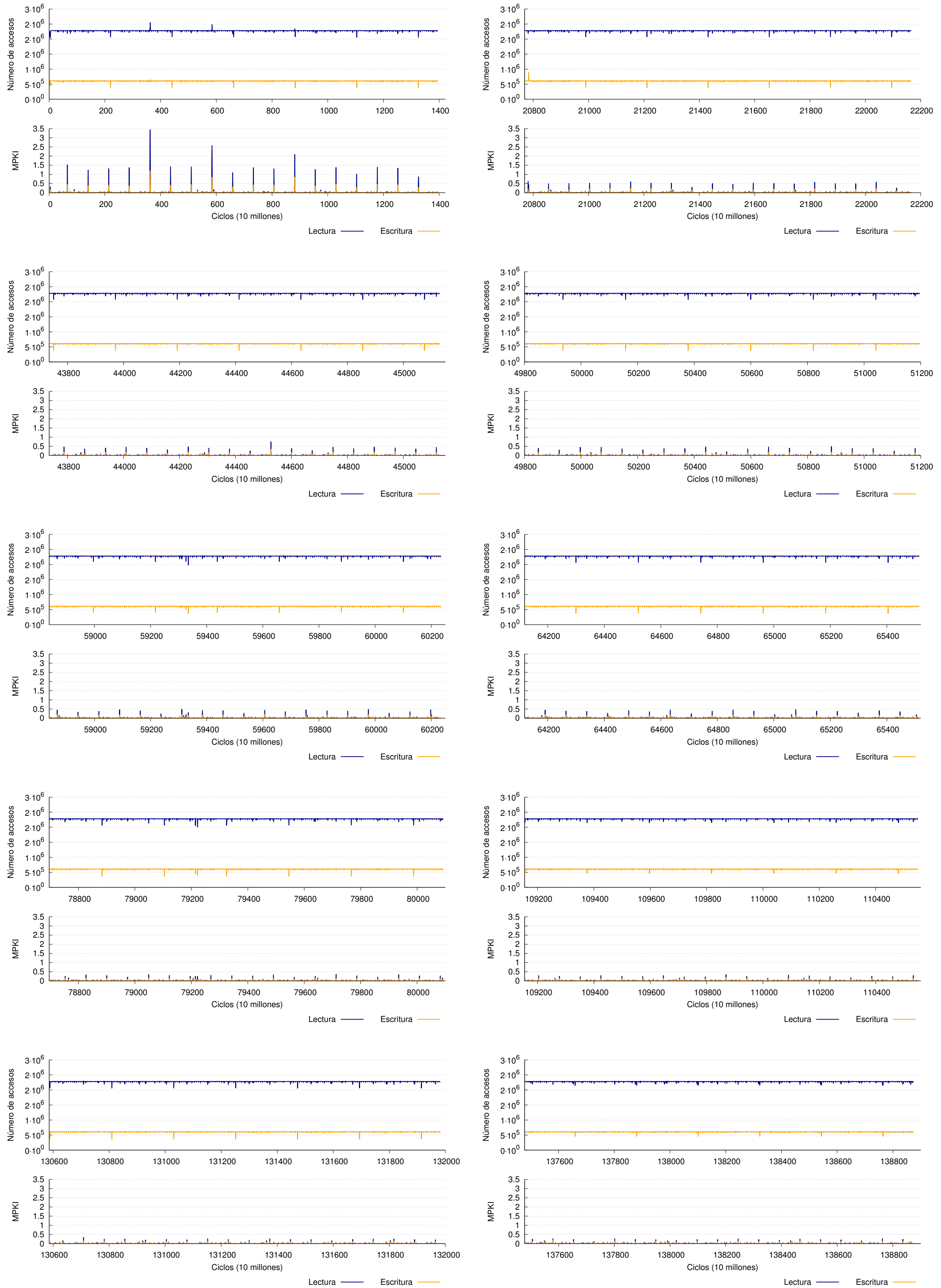


Figura D.44: Traza temporal de fallos en cache para `swaptions` con entrada nativa, ejecutado en Sparc. Aparecen diez muestras tomadas al azar del total de la ejecución. Se utiliza una política write-allocate y copy-back.

Vips

En la figura D.45 vemos que, en una arquitectura Intel, los fallos por cada mil instrucciones de la entrada nativa no son mayores que los de las entradas menores. En la mayoría de los casos, presenta un número de fallos inferior al del resto de las entradas y, a veces, es comparable a los de la entrada grande.

Para una arquitectura Sparc (figura D.46) no hemos realizado las simulaciones de la entrada nativa porque la ejecución necesitaba utilizar más espacio en disco del que disponía la arquitectura que estábamos simulando. Habría sido necesario añadir más espacio de disco y volver a comenzar el proceso de inicializar la máquina y el sistema operativo y crear los checkpoints. Decidimos, por lo tanto, que se trataba de demasiado esfuerzo para aprovecharlo luego en un único caso y que utilizaríamos únicamente los resultados de la entrada nativa para Intel. De todas formas, se ve que las relaciones entre las entradas pequeña, mediana y grande se mantienen, así que, previsiblemente, en Sparc se repetirá el mismo patrón que en Intel.

Observando las trazas temporales de la figura D.47 nos damos cuenta de que no cambia nada además de los ciclos de ejecución para cada entrada. Se mantiene un patrón de fallos prácticamente constante durante toda la ejecución, aunque parece haber menos fallos de lectura al principio y más al final (esto se nota principalmente en la entrada grande). Podemos suponer que la entrada nativa presentará un patrón de fallos similar.

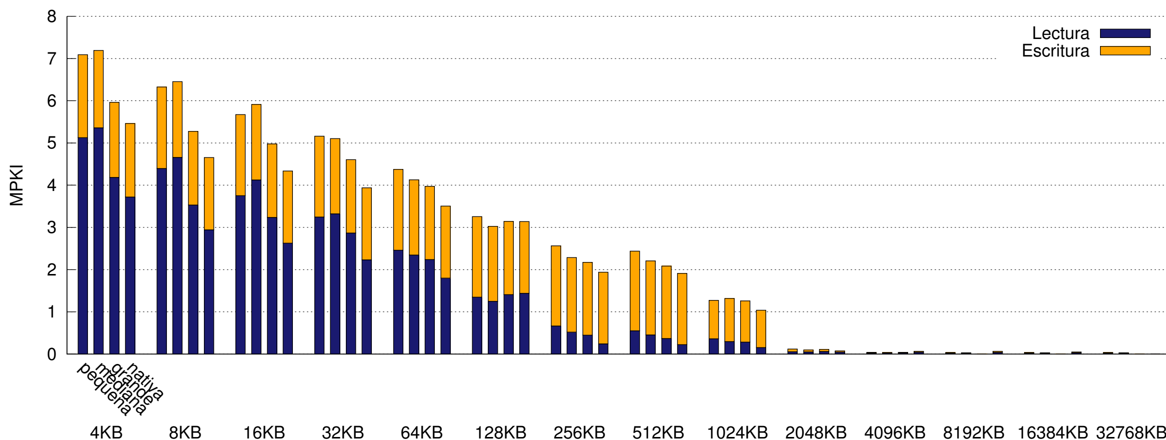
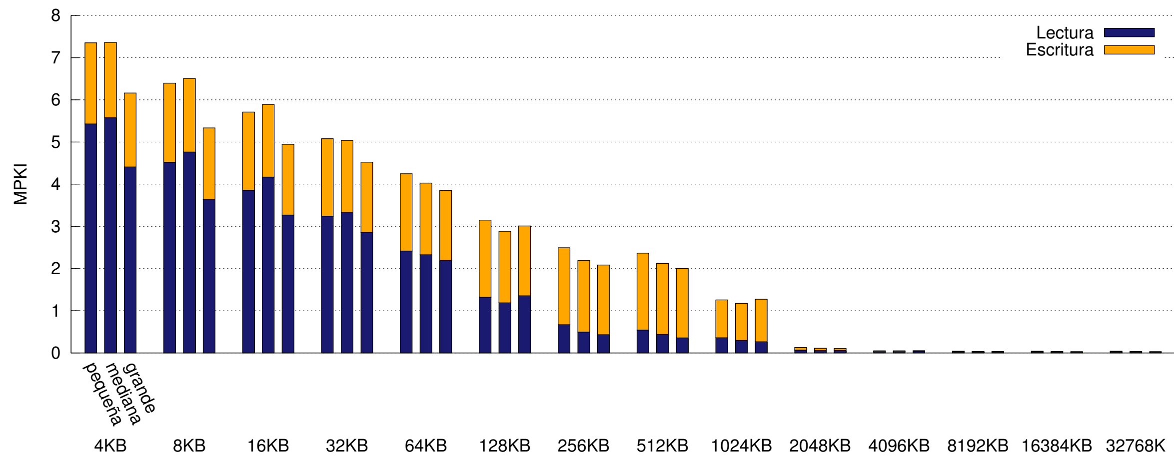
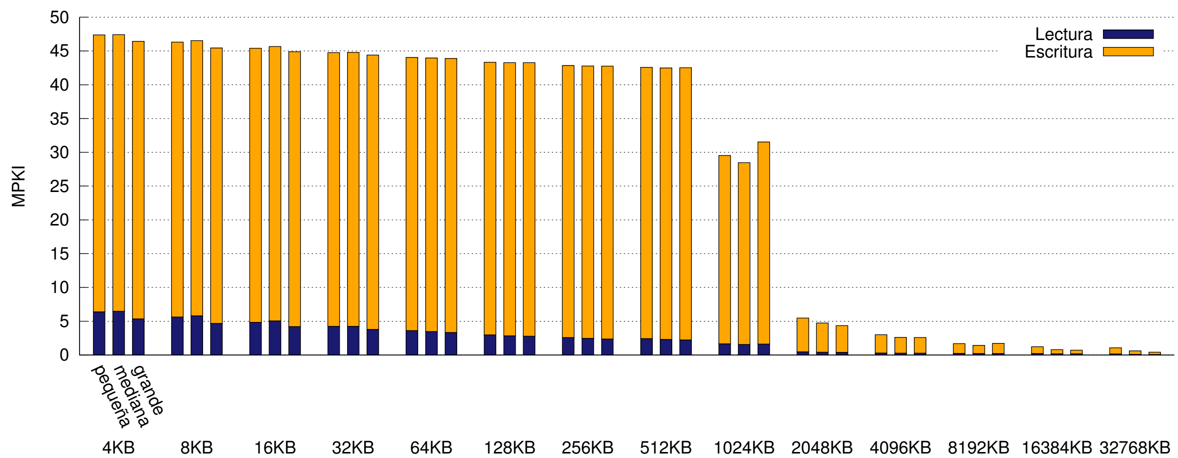


Figura D.45: Fallos por cada mil instrucciones en la cache de datos para `vips` ejecutado en Intel. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

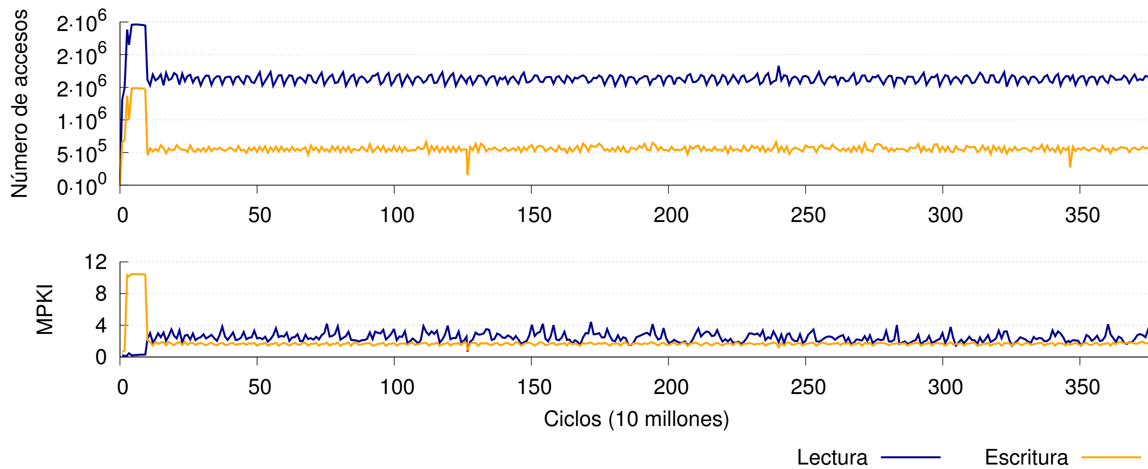


(a) write-allocate y copy-back, sólo instrucciones de usuario

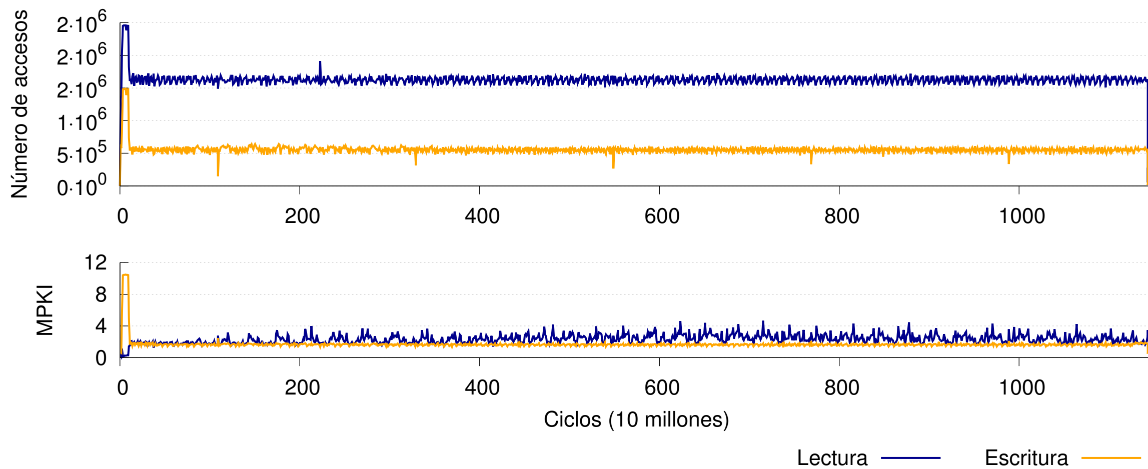


(b) non-write-allocate y write-through, instrucciones de usuario y sistema

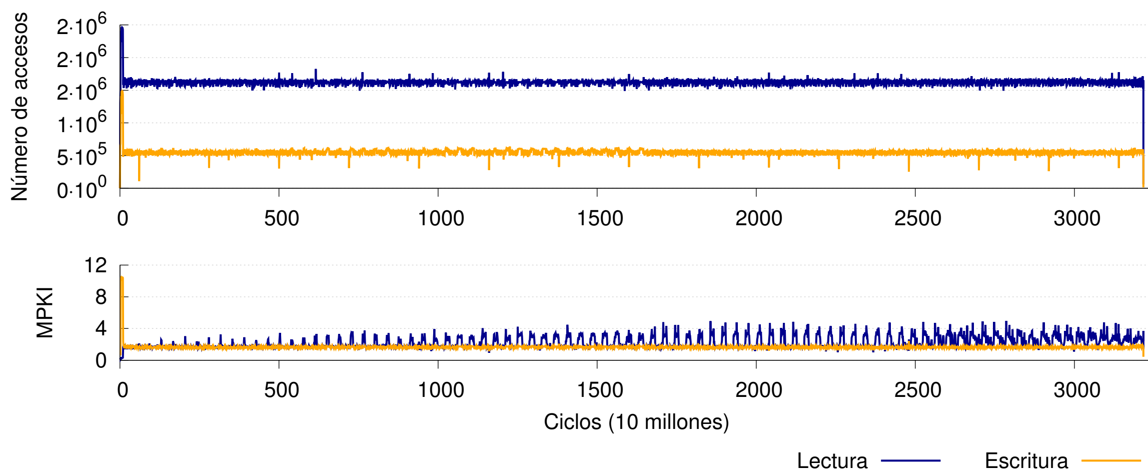
Figura D.46: Fallos por cada mil instrucciones en la cache de datos para `vips` ejecutado en Sparc.



(a) pequeña



(b) mediana



(c) grande

Figura D.47: Traza temporal de fallos en cache para `vips` con entradas pequeña, mediana y grande, ejecutado en Sparc. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

X264

En una arquitectura Intel (figura D.48), los fallos por cada mil instrucciones de la entrada nativa son siempre algo más elevados que los del resto. En Sparc (figura D.49) parece que la entrada nativa se encuentra muy próxima a la grande y la pequeña, pero tenemos poca información porque estas simulaciones resultaban extremadamente lentas debido al gran número de instrucciones ejecutadas al utilizar la entrada nativa.

En la figura D.50 podemos ver la traza temporal para las tres entradas de menor tamaño. En todos los casos hay varios picos de fallos, principalmente de escritura, pero no están distribuidos uniformemente a lo largo de la ejecución de los programas. La traza para la entrada nativa (figura D.51) sigue mostrando varios picos, aunque en este caso son menos frecuentes y tienen mayor altura.

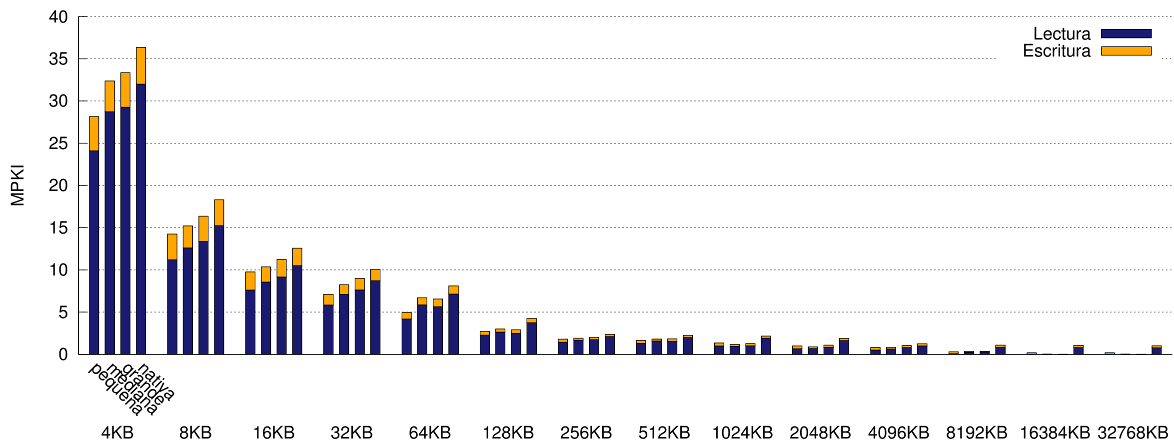
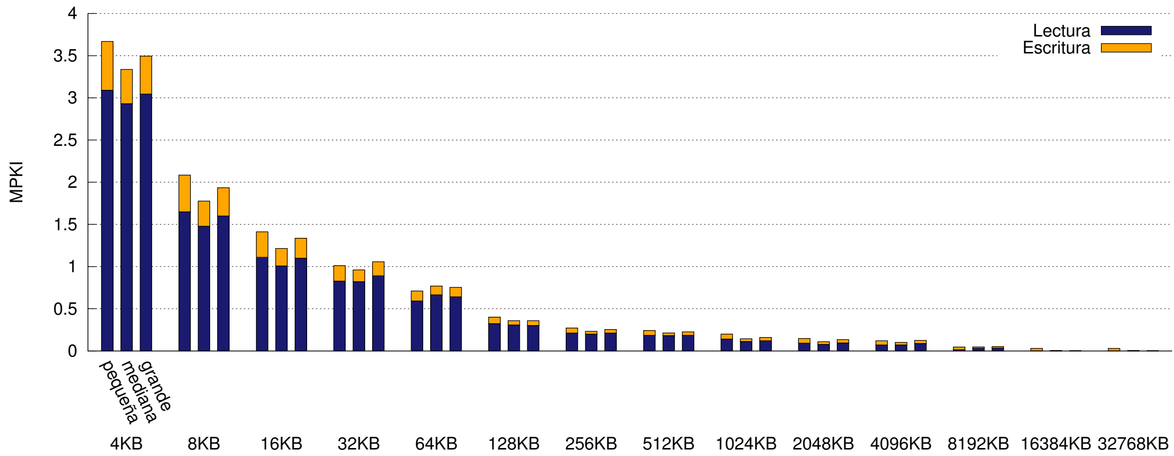
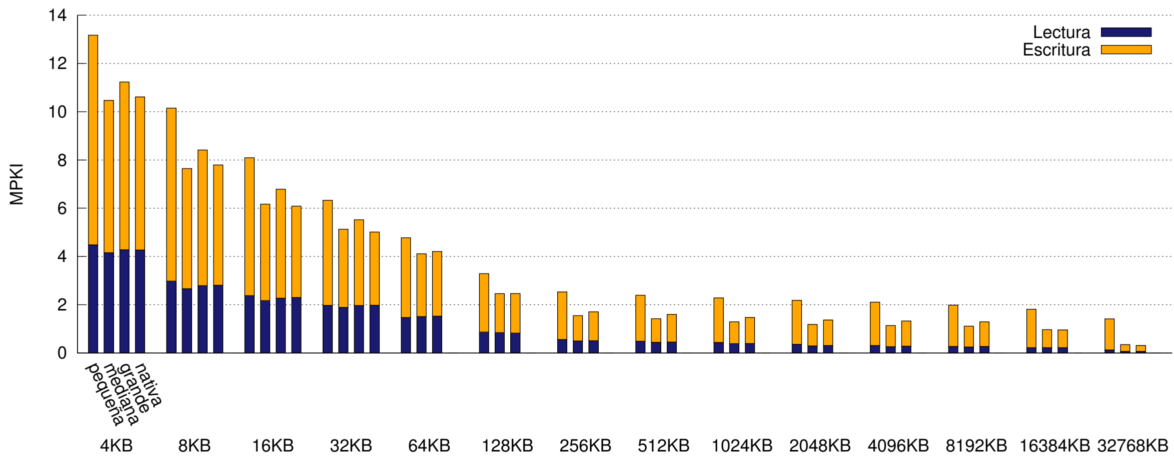


Figura D.48: Fallos por cada mil instrucciones en la cache de datos para x264 ejecutado en Intel. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

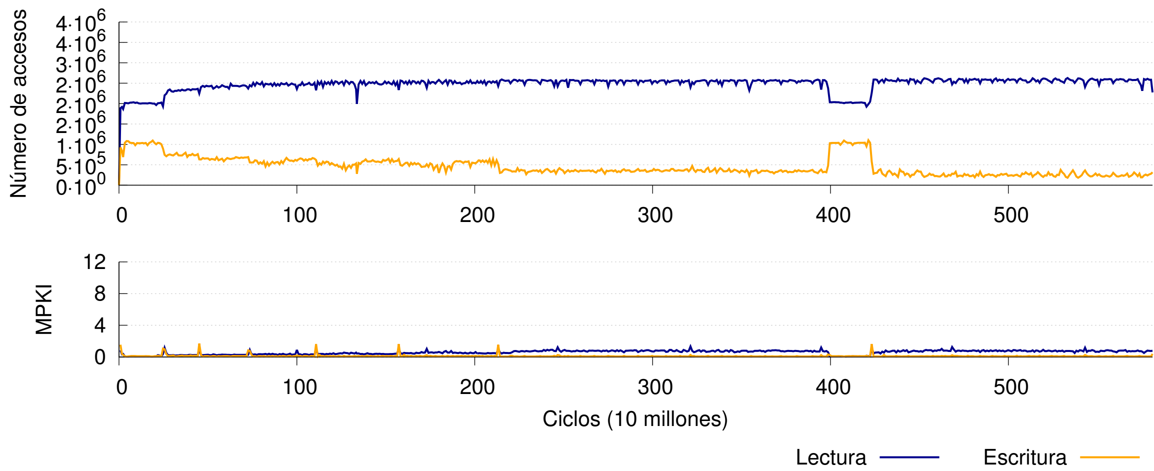


(a) write-allocate y copy-back, sólo instrucciones de usuario

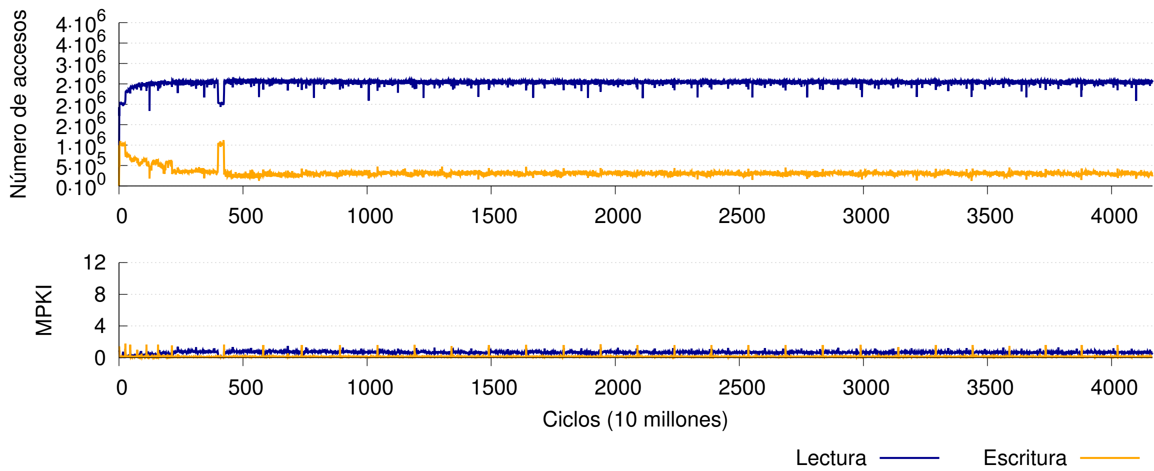


(b) non-write-allocate y write-through, instrucciones de usuario y sistema

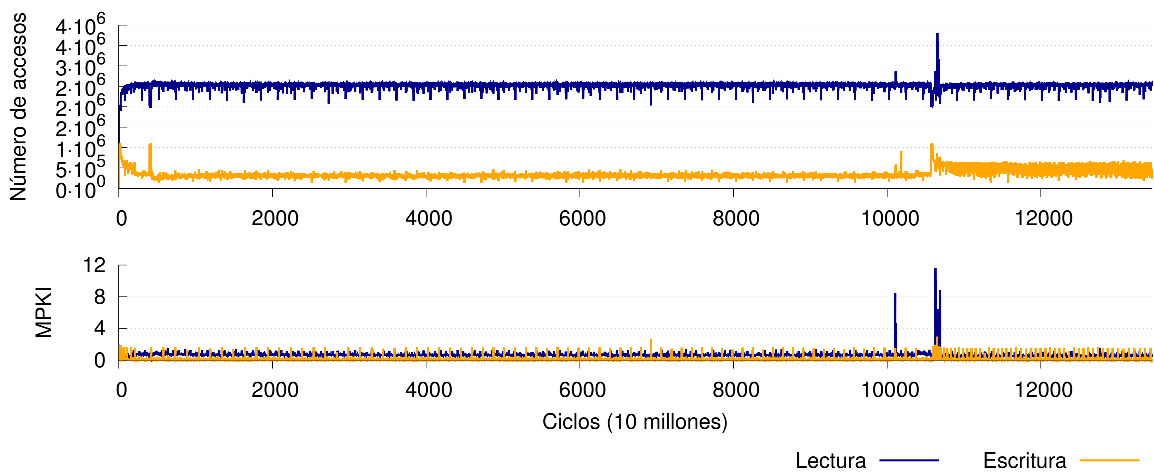
Figura D.49: Fallos por cada mil instrucciones en la cache de datos para `x264` ejecutado en Sparc.



(a) pequeña



(b) mediana



(c) grande

Figura D.50: Traza temporal de fallos en cache para `x264` con entradas pequeña, mediana y grande, ejecutado en Sparc. Se contabilizan únicamente los fallos producidos por instrucciones de usuario y se utiliza una política write-allocate y copy-back.

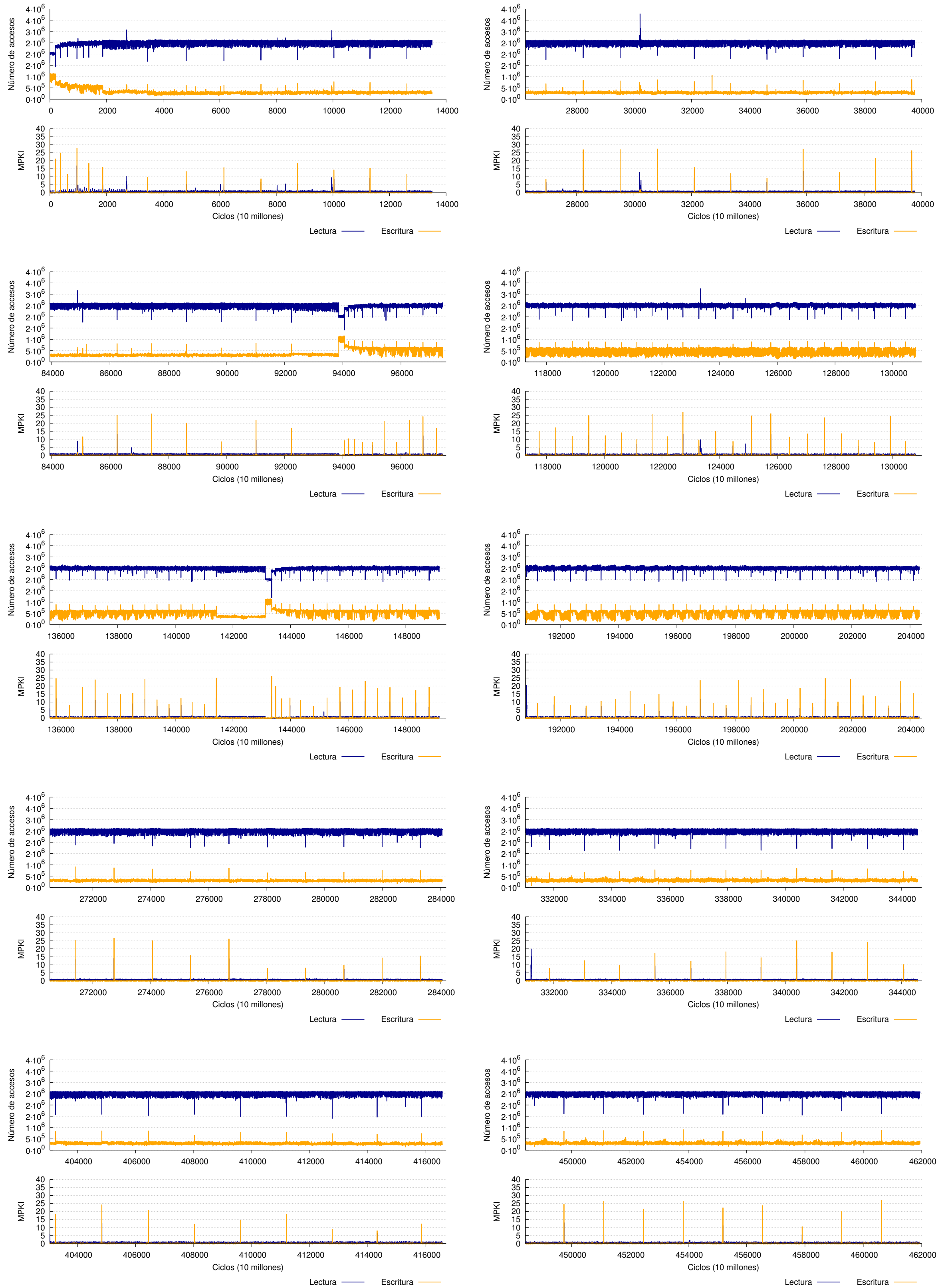


Figura D.51: Traza temporal de fallos en cache para `x264` con entrada nativa, ejecutado en Sparc. Aparecen diez muestras tomadas al azar del total de la ejecución. Se utiliza una política write-allocate y copy-back.

D.2 Selección de entradas

Considerando los resultados ya descritos en la sección D.1.4, en esta sección se propondrá lo que se considera más adecuado en cada caso para lograr una ejecución representativa de la nativa en menos tiempo o para ejercer mayor presión sobre la jerarquía de memoria. Para cada aplicación se explicará el proceso seguido para tomar la decisión.

D.2.1 Blackscholes

En la aplicación `blackscholes` veíamos que las entradas de mayor tamaño sí suponían un mayor número de fallos, especialmente a medida que aumentábamos la capacidad de la cache. Por otro lado, la traza temporal nos indicaba que el número de fallos se mantiene constante durante toda la ejecución, mostrando pequeños picos cada 750 millones de ciclos en la entrada nativa. Por lo tanto, ejecutando únicamente esa sección obtendremos el mismo resultado que al ejecutar toda la entrada nativa completa, así que bastará con ejecutar los primeros 750 millones de instrucciones de la región de interés.

D.2.2 Bodytrack

Lo más característico de este benchmark es el patrón que observamos claramente en la traza temporal. Compararemos a continuación las características de las entradas, que pueden consultarse en la tabla D.1, con la información obtenida de la traza. Por un lado, al aumentar el tamaño de la entrada aumenta el número de *frames* o fotogramas que se utilizan. Esta información se pasa al programa mediante un fichero llamado `sequenceB_x`, siendo `x` el número de frames, y con un parámetro en la línea de comandos. El fichero contiene las imágenes correspondientes al fotograma o los fotogramas que se van a analizar. Por otro lado, hay que indicar el número de partículas, que es un dato que se utiliza durante la ejecución del algoritmo. Podemos comprobar de manera directa que el número de frames se corresponde con el número de periodos que observábamos en la traza temporal. Para las entradas pequeña, mediana y grande la verificación es trivial, y para la entrada nativa obtenemos 261 frames al dividir el número total de instrucciones ejecutadas (unos 955000 millones) entre el tamaño de cada intervalo (algo menos de 4000 instrucciones). Además, la longitud del intervalo es directamente dependiente del número de partículas utilizadas. Se multiplica por dos aproximadamente al pasar de la entrada pequeña a la mediana y de la mediana a la grande, y se mantiene constante entre la grande y la nativa.

Entrada	**Número de frames**	**Número de partículas**	**Comando de ejecución**
Pequeña	1	1000	`./bodytrack input/sequenceB_1 4 1 1000 5 0 1`
Mediana	2	2000	`./bodytrack input/sequenceB_2 4 2 2000 5 0 1`
Grande	4	4000	`./bodytrack input/sequenceB_4 4 4 4000 5 0 1`
Nativa	261	4000	`./bodytrack input/sequenceB_261 4 261 4000 5 0 1`

Tabla D.1: Características de las entradas de la aplicación `bodytrack`

Teniendo en cuenta la información presentada, para conseguir una ejecución representativa de la nativa que tarde lo mínimo posible bastará con realizar los cálculos para un solo frame pero utilizando el mismo número de partículas que con la entrada nativa. De este modo estaremos ejecutando unos 4000 millones de instrucciones, tamaño comparable a la entrada mediana. Para lograr esta ejecución es suficiente con utilizar los siguientes parámetros:

```
./bodytrack input/sequenceB_1 4 1 4000 5 0 1
```

D.2.3 Canneal

En esta aplicación, las entradas de mayor tamaño presentan, en general, mayor tasa de fallos que las de menor tamaño. En la traza temporal vemos que los fallos se mantienen prácticamente constantes durante toda la ejecución, con pequeños picos de menor número de fallos en lectura. Por lo tanto, bastaría con ejecutar una parte de la entrada nativa. Siendo conservadores, podríamos simular los primeros 1500 millones de instrucciones de la región de interés de la entrada nativa.

D.2.4 Dedup

Recordamos que `dedup` presenta un número menor de fallos con su entrada grande que con el resto, y muestra también diferencias en la traza temporal debido a que el programa tiene un pipeline muy desbalanceado.

Este algoritmo construye una base de datos con todos los trozos de información únicos que encuentra en la entrada. Por lo tanto, no sólo el tamaño de la entrada alterna el uso de la jerarquía de memoria, sino los propios datos de la misma, ya que si hay menos redundancia, el tamaño de dicha base de datos será mayor.

Dadas las diferencias entre las entradas, la complejidad del algoritmo utilizado para paralelizar y la poca diferencia entre el número de instrucciones de las entradas grande y nativa, en este caso recomendamos el uso de la entrada nativa. De todas formas, será mejor utilizar versiones posteriores del benchmark en las que el pipeline esté mejor balanceado.

D.2.5 Facesim

Hemos visto ya que la tasa de fallos es igual para todas las entradas y en las trazas temporales hemos distinguido claramente un patrón. Si tomamos la entrada nativa y dividimos el tamaño del patrón (unos 230000 millones de instrucciones) entre el tamaño de la entrada completa (aproximadamente 2300000 millones de instrucciones) podemos averiguar que tenemos 100 repeticiones. Comparando esto con las características de las entradas nos damos cuenta de que la pequeña, mediana y grande constan de un solo fotograma mientras que la nativa se ocupa de 100 fotogramas, lo cual encaja perfectamente con lo que hemos observado en nuestras simulaciones. Por lo tanto, en este caso será suficiente con ejecutar la entrada pequeña para obtener un resultado representativo de la entrada nativa.

D.2.6 Ferret

Teniendo en cuenta las tasas de fallos y las trazas temporales y que, además, la simulación de la entrada nativa de esta aplicación es muy costosa en tiempo, para obtener una ejecución representativa de la nativa podemos utilizar la entrada pequeña. Si, por otro lado, nos interesa realizar una simulación que falle en cache lo máximo posible, la entrada más adecuada será la de tamaño grande.

D.2.7 Fluidanimate

De nuevo nos encontramos ante un benchmark con una traza temporal muy representativa. Vamos a contrastar las características de las entradas que presentamos en la tabla D.2 con los resultados que hemos obtenido en las simulaciones. Claramente, el número de fotogramas de la entrada se ve reflejado en el número de veces que el patrón se repite. La comprobación es trivial para las entradas pequeña, mediana y grande. Para la entrada nativa, dividiendo el número de instrucciones total de la entrada nativa (unos 2250000 millones) entre el tamaño del patrón (algo más de 4000 millones de instrucciones) obtenemos los 500 fotogramas. En cuanto a la longitud del patrón, es claramente consecuencia del número de partículas que se utilizan para modelar el fluido. Al pasar de pequeña a mediana y de mediana a grande, tanto el número de partículas como el tamaño del patrón se multiplican por tres aproximadamente. De grande a nativa, el factor de multiplicación es 1.6.

Entrada	**Número de frames**	**Número de partículas**	**Comando de ejecución**
Pequeña	5	35000	`./fluidanimate 1 5 input/in_35K.fluid out.fluid`
Mediana	5	100000	`./fluidanimate 1 5 input/in_100K.fluid out.fluid`
Grande	5	300000	`./fluidanimate 1 5 input/in_300K.fluid out.fluid`
Nativa	500	500000	`./fluidanimate 1 500 input/in_500K.fluid out.fluid`

Tabla D.2: Características de las entradas de la aplicación `fluidanimate`

Por lo tanto, la mejor opción para obtener una ejecución representativa de la nativa será ejecutar sólo 5 frames como en las entradas más pequeñas (o incluso menos, ya que para cada frame se repite el mismo patrón), pero hacerlo con 500000 partículas como con la entrada nativa. Por lo tanto, bastará con ejecutar el programa con la siguiente entrada:

```
./fluidanimate 1 5 input/in_500K.fluid out.fluid
```

D.2.8 Freqmine

En este caso, la tasa de fallos disminuye al aumentar el tamaño de la entrada y la traza temporal no nos aporta información adicional. Por lo tanto, si se desea realizar una ejecución representativa de la nativa será necesario utilizar la entrada nativa. De todas formas, nosotros recomendamos usar la entrada pequeña ya que es la que más presiona la jerarquía de memoria.

D.2.9 Raytrace

Esta aplicación presenta una tasa de fallos en cache extremadamente baja y en la traza temporal podemos distinguir claramente un patrón que se repite periódicamente.

Vamos a comparar la información que nos da la traza temporal con las características de las entradas, que se detallan en la tabla D.3. Las tres primeras entradas se diferencian únicamente en el número de píxeles, que va multiplicándose por cuatro al aumentar el tamaño de la entrada, al igual que el tiempo de ejecución. Además, en esas tres ejecuciones, se aplica el algoritmo a tres fotogramas. Si calculamos el número de instrucciones que corresponde a cada fotograma en la entrada grande, obtenemos algo menos de 2000 millones. En la entrada nativa, para la cual la

resolución es igual que en la grande, vemos claramente un patrón que se repite cada 2000 millones de instrucciones aproximadamente. Además, si dividimos el número total de instrucciones de la región de interés (algo más de 360000 millones) entre la longitud de este patrón obtenemos los 200 fotogramas que componen la entrada nativa. Las entradas grande y nativa se diferencian también en el número polígonos del objeto al que aplicamos el algoritmo, que viene indicado por la imagen que se le pasa como entrada al programa. De todas formas, parece que el único efecto que esto tiene es un aumento de la variación en el número de instrucciones de lectura ejecutadas, que es lo que nos ha permitido detectar las secciones que corresponden a los fotogramas.

Entrada	**Número de frames**	**Número de píxeles**	**Número de polígonos (millones)**	**Comando de ejecución**
Pequeña	3	480*270	1	`./rtview inputs/happy_buddha.obj -nodisplay -automove -nthreads 1 -frames 3 -res 480 270`
Mediana	3	960*540	1	`./rtview inputs/happy_buddha.obj -nodisplay -automove -nthreads 1 -frames 3 -res 960 540`
Grande	3	1920*1080	1	`./rtview inputs/happy_buddha.obj -nodisplay -automove -nthreads 1 -frames 3 -res 1920 1080`
Nativa	200	1920*1080	10	`./rtview inputs/thai_statue.obj -nodisplay -automove -nthreads 1 -frames 200 -res 1920 1080`

Tabla D.3: Características de las entradas de la aplicación `raytrace`

Por lo tanto, para obtener una ejecución representativa de la nativa sólo tendremos que ejecutar tres fotogramas al igual que en las entradas de menor tamaño (siendo conservadores, ya que se repetirá el mismo patrón tres veces), pero hacerlo con la resolución y número de polígonos de la entrada nativa. Simplemente tendremos que utilizar los siguientes parámetros:

```
./rtview inputs/thai_statue.obj -nodisplay -automove -nthreads 1 -frames 3 -res
1920 1080
```

A pesar de todo, insistimos en que los fallos en cache son prácticamente insignificantes y el benchmark no será adecuado para un estudio de las prestaciones de la jerarquía de memoria.

D.2.10 Streamcluster

En esta aplicación, las entradas de mayor tamaño suponen un mayor número de fallos y la traza temporal muestra que la tasa de fallos se mantiene constante durante toda la ejecución, a excepción de la zona inicial. Por lo tanto, bastará con ejecutar una sección de la entrada nativa. Será suficiente con 10000 millones de instrucciones, pero saltando los 5000 millones de instrucciones iniciales para asegurarnos de que no medimos la zona inicial en la que todavía no se ha estabilizado el número de fallos. Además, será importante calentar la cache antes de tomar ninguna estadística, tal y como se explicó en la sección 4.5.

D.2.11 Swaptions

En esta aplicación, la tasa de fallos toma valores excesivamente pequeños, además de no presentar ninguna variación al aumentar el tamaño de la entrada. La traza temporal muestra cómo los fallos se mantienen constantes a lo largo de la ejecución de todas las entradas, con pequeños picos que se repiten periódicamente.

En la tabla D.4 se muestran las características de la entrada junto con un valor que representa el tamaño total de la entrada, que ha sido obtenido multiplicando el número de `swaptions` por el de simulaciones. Si comparamos este valor con el número de ciclos que tarda en ejecutarse el benchmark (que recordamos que es igual al número de instrucciones ya que nuestras simulaciones tienen un IPC de 1), vemos que en ambos casos se aplica un factor multiplicativo de cuatro al pasar de pequeña a mediana y de mediana a grande, y de 100 al pasar de grande a nativa. Por lo tanto, vemos que el único efecto que tienen los parámetros de entrada es alargar el tiempo de ejecución.

Entrada	**Número de `swaptions`**	**Número de simulaciones**	**Tamaño de la entrada**
Pequeña	16	5000	16 x 5000 = 80000
Mediana	32	10000	32 x 10000 = 320000
Grande	64	20000	64 x 20000 = 1280000
Nativa	128	1000000	128 x 1000000 = 128000000

Tabla D.4: Características de las entradas de la aplicación `swaptions`

Para lograr una ejecución representativa de la entrada nativa será suficiente con usar la entrada pequeña, aunque recordamos que el número de fallos de cache de esta aplicación es muy pequeño.

D.2.12 Vips

Para esta aplicación la entrada pequeña presenta, en general, más fallos en cache. Las trazas temporales tienen el mismo aspecto en las entradas pequeña, mediana y grande, aunque recordamos que no se dispone de resultados para la entrada nativa.

La única característica que diferencia unas entradas de otras es la resolución de la imagen que procesará el algoritmo, que tiene un impacto directo en el número de instrucciones. Se prevé que la entrada nativa tendrá una traza similar al resto y que será posible obtener resultados representativos simulando únicamente una sección. De todas formas, para no tomar la decisión basándonos en suposiciones podemos simplemente ejecutar la entrada pequeña, que es la más adecuada para realizar un estudio de la jerarquía de memoria porque es la que más fallos por cada mil instrucciones presenta.

D.2.13 X264

En este caso, la tasa de fallos para la entrada nativa es mayor que para el resto en una arquitectura Intel, pero se encuentra muy próxima a la grande y la pequeña en Sparc. Las trazas temporales

muestran varios picos de fallos, aunque no siempre están distribuidos uniformemente a lo largo de la ejecución de las aplicaciones.

Las entradas pequeña, mediana y grande se diferencian únicamente en el número de fotogramas de video que se deben codificar, lo cual tiene una relación directa con el número de picos que se observan en la traza temporal. En la traza para la entrada nativa, el número de picos también se corresponde con el número de fotogramas, pero en este caso aumenta también la resolución de los fotogramas. Esto tiene repercusión tanto en el tiempo que tarda en procesarse cada fotograma como en la altura de los picos de fallos, que ahora es mucho mayor. En este algoritmo no resulta tan sencillo como en casos anteriores (`bodytrack`, `fluidanimate` y `raytrace`) utilizar una entrada nueva combinando los parámetros de varias entradas, ya que toda la información se pasa en el fichero de vídeo, así que habría que preparar uno nuevo.

El algoritmo de este benchmark tiene que utilizar para algunos fotogramas la información de otros fotogramas ya procesados, lo cual explica por qué el tiempo y los fallos no se mantienen constantes para cada fotograma. Además, la resolución afecta en mayor medida porque no sólo implica que el fotograma estudiado es más grande, si no que, en caso de tener que acceder a otros fotogramas, se deberá acceder a cantidades de datos más grandes. Por último, los propios datos, no sólo su tamaño, influyen también en cómo se ejecutará el algoritmo, ya que dependiendo del vídeo que se desee codificar cambiará la cantidad de veces que será necesario acceder a fotogramas anteriores.

Por lo tanto, la selección de una entrada representativa de una ejecución nativa es especialmente complicada en este caso. El mejor modo de lograr un resultado válido será simular varias zonas distintas de la entrada nativa, calentando la cache antes de cada una. Será suficiente con tomar cuatro muestras de 20000 millones de ciclos cada una en puntos aleatorios de la ejecución de la aplicación. Así se tendrán en cuenta zonas en las que el procesado de los fotogramas es más lento y zonas en las que es más rápido.

Glosario

Benchmark Carga de trabajo artificial que incluye las características más importantes de cargas de trabajo reales y relevantes. En general, son aplicaciones pequeñas, eficientes y controlables.

Checkpoint Estado de una simulación que almacenamos para poder volver al mismo punto rápidamente más tarde.

Copy-back Política de escritura en la cual las escrituras sólo se llevan a cabo en la cache y se escribirán en la memoria principal cuando el bloque sea reemplazado.

Footprint El *footprint* o huella es el número total de páginas de memoria a las que un programa accede cuando es ejecutado.

Host En una simulación, el *host* es la máquina sobre la que ejecutamos la simulación.

Instruction mix Número de instrucciones de cada tipo que hay en un programa, ya sean aritmético-lógicas, de memoria,...

Muestreo basado en eventos El muestreo basado en eventos (en inglés, *event based sampling* o EBS) es un método utilizado en *profiling* que se basa en interrumpir la ejecución de la aplicación cada cierto número de eventos y anotar en qué punto del código se encuentra. De esta manera se obtiene un histograma del número eventos basado en las líneas de código en que se producen.

Muestreo basado en tiempo El muestreo basado en tiempo (en inglés, *time based sampling* o TBS) es un método utilizado en *profiling* que se basa en interrumpir la ejecución de la aplicación cada cierto tiempo y anotar en qué punto del código se encuentra. De esta manera se puede conocer en qué zonas del código ha pasado más tiempo la ejecución.

Non-write-allocate Política de escritura en la cual, ante un fallo en escritura, el bloque se modifica en memoria principal y no se trae a la cache.

Pipeline parallelism *Pipelining* es un modelo de programación utilizado para explotar el paralelismo a nivel de tarea. El trabajo a realizar se divide en varias etapas que se ejecutarán concurrentemente en un multiprocesador. Las etapas del pipeline tienen una relación productor-consumidor e intercambian información mediante colas. Dependiendo del diseño, uno o más threads pueden encargarse de cada etapa.

Profiling El profiling es una técnica que permite inspeccionar el funcionamiento interno de una aplicación durante su tiempo de ejecución.

Región de interés La región de interés o ROI (del inglés *Region of Interest*) es la parte de una aplicación que resulta relevante, quedando fuera la inicialización en la se cargan los datos a utilizar y el final en el que se escribe el resultado.

Simulador de sistema completo Simulador que incluye procesadores, memoria, interfaces de red y otros periféricos. Se utiliza para el diseño, desarrollo y prueba de hardware y software en un entorno que se aproxima al contexto final de aplicación del producto.

Slowdown Es la medida de cuántas veces más lenta resulta la simulación de la aplicación respecto de su ejecución nativa.

Target En una simulación, el *target* es el sistema que estamos simulando.

Thread pool Mediante este método, un thread principal se ocupa de ir distribuyendo el trabajo entre los threads disponibles. Permite que un algoritmo reutilice los threads para eliminar la necesidad de destruirlos y crear otros nuevos.

Working set Es el conjunto de la páginas que un proceso utiliza en un determinado intervalo de tiempo

Write-allocate Política de escritura en la cual, ante un fallo en escritura, el bloque correspondiente se trae a la cache.

Write-through Política de escritura en la cual cada escritura en la cache general una escritura también en memoria principal.