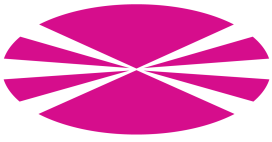
PhD Thesis

Cache Design Strategies for Efficient Adaptive Line Placement

Dyer Rolán García

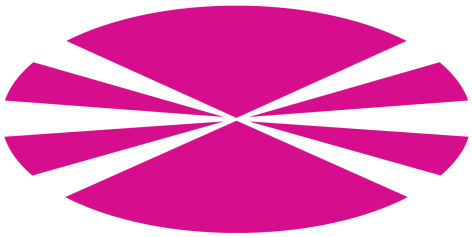
2012



Computer Architecture Group

Universidade da Coruña, Spain

Computer Architecture Group
Universidade da Coruña, Spain



PhD Thesis

Cache Design Strategies for Efficient Adaptive Line Placement

Dyer Rolán García

April 2012

PhD Advisors:
Basilio B. Fraguela Rodríguez
Ramón Doallo Biempica

Dr. Basilio B. Fraguela Rodríguez
Titular de Universidad
Dpto. de Electrónica y Sistemas
Universidade da Coruña

Dr. Ramón Doallo Biempica
Catedrático de Universidad
Dpto. de Electrónica y Sistemas
Universidade da Coruña

CERTIFICAN

Que la memoria titulada "*Cache Design Strategies for Efficient Adaptive Line Placement*" ha sido realizada por D. Dyer Rolán García bajo nuestra dirección en el Departamento de Electrónica y Sistemas de la Universidade da Coruña y concluye la Tesis Doctoral que presenta para optar al grado de Doctor en Ingeniería Informática con la Mención de Doctor Internacional.

En A Coruña, a 27 de Marzo de 2012

Fdo.: Basilio B. Fraguela Rodríguez
Director de la Tesis Doctoral

Fdo.: Ramón Doallo Biempica
Director de la Tesis Doctoral

Vº Bº: Juan Touriño Domínguez
Director del Dpto. de Electrónica y Sistemas

The Dissertation Committee for Dyer Rolán García certifies that this is the approved version of the following dissertation:

Cache Design Strategies for Efficient Adaptive Line Placement

Committee:

_____

President,

_____

Member,

_____

Member,

_____

Member,

_____

Secretary,

Resumen

Introducción

Las memorias caché, o simplemente cachés, representan un papel crucial en el rendimiento de las computadoras salvando la diferencia de velocidad que existe entre los procesadores y la memoria principal. La gestión de memorias caché ha adquirido aún más relevancia debido a la aparición de los procesadores multinúcleo (CMPs), que suponen requisitos de ancho de banda superiores, mayores conjuntos de trabajo de aplicaciones emergentes y que además requieren una distribución eficiente de los recursos caché entre todos los núcleos de procesamiento.

Esta Tesis se centra en analizar algunos de los problemas que se encuentran habitualmente en las cachés modernas y propone soluciones efectivas y económicas para mejorar su rendimiento. La mayoría de los diseños propuestos en esta Tesis son capaces de reducir la tasa de fallos basándose en los diferentes niveles de demanda de cada conjunto caché. De este modo, las líneas caché se ubican en bloques infrautilizados de otros conjuntos caché si es probable que vuelvan a ser referenciadas y su conjunto nativo está experimentando problemas de capacidad. Cuando esto no es suficiente, esta Tesis propone modificar de un modo coordinado la política de inserción de estos conjuntos caché. Por tanto, nuestras propuestas retienen la mayor parte útil del conjunto de trabajo a la vez que descartan datos temporales tan pronto como sea posible. Estas ideas, inicialmente aplicadas en cachés de último nivel, en inglés Last-Level Caches (LLCs), de monoprocesadores, son adaptadas satisfactoriamente en esta Tesis a cachés de primer nivel y sistemas multinúcleo. En cuanto a las cachés de primer nivel, proponemos un diseño novedoso que permite la distribución de recursos caché entre instrucciones y datos dependiendo de sus

necesidades particulares. A continuación, en sistemas multinúcleo, nuestros diseños son en un primer momento mejorados mediante la inclusión de soporte para tratar cada hilo que comparte recursos en una caché compartida de un modo particular para, a continuación, proponer una política de inserción específicamente diseñada para este entorno. Finalmente, exploramos la compartición de recursos en CMPs con LLCs privadas mediante el desplazamiento de líneas entre las distintas cachés que constituyen el último nivel de la jerarquía. Este último diseño incluye además varios puntos novedosos como la inclusión de un estado neutral en el mecanismo de desplazamiento que impide que un conjunto caché tome parte en el mismo si esto pudiese llegar a ser perjudicial, el uso de distintas granularidades para la gestión de la caché o la aplicación coordinada de la política de inserción más conveniente con los mecanismos previos. A lo largo de todo este proceso hemos usado una métrica sencilla y efectiva para determinar el estado, o nivel de saturación, de los conjuntos caché llamada Set Saturation Level (SSL). Finalmente, cabe destacar que nuestros diseños se han mostrado muy competitivos con respecto a otras propuestas recientes, incluso a menudo superándolas, a pesar de suponer costes de almacenamiento y consumo de energía casi insignificantes.

Metodología

El desarrollo de esta Tesis sigue las líneas de metodologías clásicas; incluyendo planificación, diseño, análisis de costes, evaluación y viabilidad de las implementaciones propuestas. El principal objetivo de esta Tesis es enriquecer el diseño moderno de memorias caché mediante la propuesta de nuevas políticas para monoprocesadores y su posterior adaptación a procesadores multinúcleo. Veremos que si bien los diseños para los primeros son directamente aplicables a los segundos, estos diseños deben adaptarse si queremos alcanzar el mejor nivel de prestaciones en los procesadores multinúcleo.

En primer lugar, se desarrolló un estudio cuidadoso del estado del arte en el campo de las memorias caché, inicialmente en el último nivel de la jerarquía. Como resultado, se detectaron y analizaron comportamientos perjudiciales y debilidades en diseños previos y se propusieron soluciones eficientes. En segundo lugar, se realizó la configuración de un entorno adecuado para la evaluación de las soluciones propues-

tas. A continuación, se depuraron y ajustaron los diseños propuestos para obtener un rendimiento óptimo con un coste adicional poco significativo. Estos diseños previos, que se evaluaron exitosamente en entornos monoprocesador, fueron sucesivamente adaptados a entornos con procesadores multinúcleo siguiendo las mismas fases.

Finalmente, todos estos esfuerzos han contribuido a lograr el objetivo principal de mejorar el rendimiento de las memorias caché manteniendo el mismo grado de complejidad que en el funcionamiento caché tradicional, bajo consumo de potencia y costes de almacenamiento adicionales mínimos. De hecho estos diseños han superado a las técnicas más recientes en este campo, tanto para entornos monoprocesador como para procesadores multinúcleo.

Conclusiones

A fin de neutralizar la diferencia de velocidad entre el procesador y la memoria principal, los procesadores modernos cuentan con grandes cachés de último nivel (LLC). Sin embargo, los diseños tradicionales, originalmente desarrollados para pequeñas cachés de primer nivel, son ineficientes para cachés mayores y LLCs compartidas en procesadores multinúcleo. La importancia de la gestión de la caché se ha vuelto más crítica debido al aumento de los requisitos de ancho de banda de los CMPs, el incremento de los conjuntos de trabajo de aplicaciones emergentes, y el menor espacio de caché dedicado a cada procesador debido al aumento del número de procesadores por chip. Esta Tesis, *"Cache Design Strategies for Efficient Adaptive Line Placement"*, analiza algunos de los problemas que aparecen al gestionar grandes cachés y los comportamientos perjudiciales que experimentan para así proponer soluciones efectivas y aumentar su rendimiento. De hecho, esta Tesis demuestra que cambios sencillos y con un coste adicional mínimo pueden incrementar substancialmente el rendimiento de estas cachés. Una característica común a todos los diseños propuestos en esta Tesis es que analizan y aplican políticas de grano fino para mejorar el rendimiento sin que ello implique un coste de almacenamiento elevado ni grandes modificaciones en el funcionamiento habitual de la caché.

Esta Tesis demuestra que los efectos perjudiciales derivados de una de las ineficiencias más comunes en las memorias caché, la no uniformidad de las referencias a

memoria sobre los distintos conjuntos caché, puede reducirse mediante el desplazamiento de líneas desde conjuntos que requieren más espacio hacia otros que pueden proporcionarlo mediante el establecimiento de asociaciones entre ellos. La Set Balancing Cache (SBC) [60] asocia estas dos clases de conjuntos con el fin de equilibrar la carga de datos entre ellos. Para determinar el estado de cada conjunto la SBC utiliza una métrica sencilla y efectiva denominada Set Saturation Level (SSL), que mide hasta qué punto es capaz un conjunto dado de la caché de retener su conjunto de trabajo particular. Este valor se calcula para cada conjunto gracias a un contador con aritmética de saturación que se incrementa con cada fallo caché que se produce en el conjunto y se reduce con cada acierto. Esta métrica ha demostrado ser capaz de establecer el rol más adecuado de cada conjunto dentro de una asociación. Un diseño estático inicial (SSBC), el cual permite únicamente desplazamientos entre parejas de conjuntos preestablecidas, logró una reducción media en la tasa de fallos de 9.2 %, o 14 % si se calcula en términos de media geométrica. Esta reducción resultó en una mejora media en el IPC entre 2.7 % y 3.7 % dependiendo de la configuración usada. Además, se propone una estructura sencilla y económica, llamada Destination Set Selector (DSS), capaz de proporcionar el conjunto libre dentro de la caché más conveniente para recibir líneas ante una solicitud de asociación. Una versión dinámica mejorada de la SBC (DSBC), la cual asocia conjuntos caché altamente saturados con aquellos menos saturados gracias al DSS, obtuvo una reducción media en la tasa de fallos de 12.8 %, 19 % computado como la media geométrica, consiguiendo una mejora en el IPC de entre 3.5 % y 5.25 % dependiendo de la jerarquía de memoria considerada. Finalmente, los distintos diseños de la SBC demostraron consistentemente ser mejores que sucesivos incrementos en la asociatividad, tanto en términos de área requerida como rendimiento, además de suponer costes de almacenamiento mínimos; menos de un 0.6 % con respecto a la configuración base.

Esta Tesis confirma además que otro de los problemas más habituales en las memorias caché, el *thrashing*, puede aliviarse mediante la aplicación de una política de inserción capaz de descartar datos temporales lo antes posible a la vez que mantiene la sección con más localidad del conjunto de trabajo en la caché. Este tipo de política puede combinarse con la DSBC propuesta en esta Tesis para reducir la presión sobre los conjuntos caché cuando el desplazamiento de líneas entre los mismos no es suficiente. Inicialmente se analizan las razones que llevan a un comportamiento no óptimo de la combinación entre la DSBC y una política de inserción

específicamente diseñada para atacar problemas de capacidad como es la Dynamic Insertion Policy (DIP) [53] en una caché. En base a ello, se propone de un modo razonado un diseño integrado de ambas políticas que permite su cooperación efectiva: la Bimodal Set Balancing Cache (BSBC) [61]. Este diseño extendido trata con fallos de conflicto y de capacidad mediante el uso del Set Saturation Level como único árbitro de control de ambos comportamientos. Por tanto, esta Tesis demuestra además la utilidad del SSL para detectar problemas de capacidad globales en la caché así como desequilibrios entre conjuntos. De este modo, la BSBC implica sólo un 0.6 % de coste de almacenamiento adicional con respecto a la caché base. Experimentos usando benchmarks con características variadas muestran que la aplicación conjunta de la DSBC y DIP en una caché puede resultar un diseño poco afortunado si no se coordina adecuadamente, o que, por el contrario, puede proporcionar los mejores resultados cuando se integran apropiadamente en nuestro diseño BSBC. Por ejemplo, en una caché de segundo nivel de 2MB y 8 vías, la combinación sin coordinación de ambas propuestas, DIP+DSBC, obtiene una reducción relativa en la tasa de fallos de 8.3 %, mientras que la DSBC y DIP de un modo aislado la reducen en un 12 % y un 10 % respectivamente. Con la BSBC la reducción asciende hasta el 16 %. Como resultado, la BSBC consigue las mayores mejoras en términos de IPC, 4.8 % para la misma configuración, en comparación con el 3 % que proporciona la combinación sin coordinación DSBC+DIP. El resto de políticas probadas; DSBC, DIP y la probabilistic escape LIFO, obtienen resultados intermedios. Por tanto, esta Tesis muestra que la coordinación de distintas políticas para tratar distintos problemas puede gestionarse con sencillas y económicas métricas y proporcionar beneficios mucho mayores que la aplicación independiente de dichas políticas. Adicionalmente, hemos demostrado que la BSBC es directamente aplicable a una caché compartida, donde se ha desenvuelto favorablemente, logrando una reducción media del 10 % en la tasa de fallos y una mejora en el IPC del 3 %, superando incluso a técnicas específicamente diseñadas para rendir en este tipo de entorno como PIPP [84]. También se ha evaluado la DSBC en una caché compartida, reduciendo la tasa de fallos en un 7.8 % y logrando una mejora en la productividad de hasta un 10 %. A pesar de estos resultados positivos, técnicas que incluyen soporte para tratar a cada hilo de un modo independiente, como TADIP [27], han obtenido mejores resultados a medida que se incrementa el número de procesadores que comparten los recursos. Esto sugiere la importancia de dar un tratamiento particular a cada flujo de acceso

independiente en la caché.

A continuación esta Tesis muestra que la idea de dar tratamientos particulares a distintos flujos de acceso de acuerdo a su comportamiento individual puede aplicarse satisfactoriamente a cachés de primer nivel mediante el ajuste dinámico de los recursos destinados a instrucciones y datos, los dos principales flujos de acceso que existen a este nivel de la jerarquía, dependiendo de su demanda particular. Proponemos la Virtually Split Cache (VSC), el primer diseño que es consciente de la distinta localidad que tienen las instrucciones con respecto a los datos y que reserva recursos, específicamente bancos de memoria, para ambos tipos de información dependiendo de la demanda de cada uno. Hemos propuesto dos diseños alternativos para determinar los recursos que demandan tanto instrucciones como datos. El primer diseño, la Shadow Tag VSC, usa etiquetas extra para decidir si asignar un banco de memoria adicional para instrucciones, o datos, aumenta el rendimiento. El segundo diseño, la Global Selector VSC, usa un contador común con aritmética de saturación para que instrucciones y datos se batan en duelo por los recursos. La Shadow Tag VSC logró un 3.7 % de mejora en el IPC, una reducción media en la tasa de fallos del 13 % y una reducción en el consumo de potencia de la jerarquía de memoria del 10 %, con respecto a un diseño caché separado para instrucciones y datos. La Global Selector VSC obtuvo un 3.2 % de mejora en el IPC y una reducción media en la tasa de fallos del 11 %, necesitando sólo 4 bits de almacenamiento adicional, mientras que redujo el consumo de potencia en un 8 %. Además, ambos diseños demostraron un buen desempeño en entornos multinúcleo. Así, la Shadow Tag VSC y la Global Selector VSC mejoraron una configuración base con 4 procesadores en términos de productividad en un 4.5 % y un 3.7 % de media, respectivamente.

Más adelante esta Tesis se centra en LLCs compartidas, donde pueden encontrarse también los comportamientos analizados previamente. La Thread-Aware Bimodal Set Balancing Cache (TABSBC) mide el nivel de presión que cada aplicación ejerce en cada conjunto de la caché utilizando el Set Saturation Level. Este diseño incluye una nueva política de inserción, llamada BIP-C, específicamente diseñada para reducir problemas de capacidad en cachés compartidas. Esta política supone una mejora considerable con respecto a BIP [53] en estas cachés, puesto que protege a las líneas de ser desalojadas debido a accesos de otros procesadores. Cuando la TABSBC estima que una aplicación está experimentando un mal comportamiento

caché, trata en un primer momento de desplazar las líneas de la aplicación problemática hacia conjuntos caché con espacio libre aplicando técnicas basadas en la Set Balancing Cache. Cuando esto no es posible o suficiente, recurre a BIP-C para la aplicación y el conjunto caché en cuestión. La TABSBC proporciona de un modo sensato y coordinado un mecanismo capaz de aplicar sus políticas subyacentes considerando los distintos hilos que comparten los recursos. A pesar de su naturaleza de grano fino su coste de almacenamiento es muy razonable, sobre un 1 % o incluso menos en configuraciones representativas. Una extensa experimentación usando una amplia gama de benchmarks indica que la TABSBC consigue consistentemente los mejores resultados en comparación con propuestas recientes. Esto se debe a dos características principales que la distinguen de otras propuestas. La primera es la capacidad de aplicar políticas de grano fino en contraposición a las políticas globales que usan otras técnicas. La segunda es la coordinación de mecanismos capaces de reducir fallos de conflicto y de capacidad. Este último detalle es ignorado por otras propuestas específicamente diseñadas para cachés compartidas. Por estas mismas razones, la TABSBC sigue manteniendo sus buenos resultados a medida que se aumenta el número de procesadores que comparten la caché.

Finalmente, esta Tesis reafirma el hecho de que un modo habitual de incrementar el rendimiento en CMPs con LLCs privadas es proporcionar un mecanismo que habilite la compartición de recursos mediante el desplazamiento de líneas. Proponemos el Adaptive Set-Granular Cooperative Caching (ASCC) [62], un diseño capaz de determinar el estado de cada conjunto caché y aplicar las políticas más adecuadas a cada uno, en comparación con otros diseños que aplican políticas globalmente. Este diseño realiza desplazamientos entre conjuntos de distintas cachés y aplica la política de inserción más conveniente a cada conjunto si los desplazamientos no son suficientes para reducir los fallos de capacidad, apoyándose en el SSL. Proponemos una nueva política de inserción, SABIP, específicamente diseñada para abordar problemas de capacidad en entornos donde se realizan desplazamientos entre cachés. Además, introducimos un estado neutral para conjuntos individuales dentro del mecanismo de desplazamientos, de forma que éstos no participan ni como emisores ni como receptores de líneas en los mismos. Los beneficios de mantener partes de la caché en este estado, es decir, sin tomar parte en el mecanismo de desplazamientos, han demostrado ser cuantiosos. ASCC logró una mejora en el rendimiento del 6.4 % y del 5.7 % ejecutando 2 y 4 aplicaciones, respectivamente, lo que se tradujo

en una reducción de la latencia media de memoria del 18 % y 21 %. Su coste de almacenamiento ha sido estimado en un 0.17 % con respecto a la configuración base. Además, esta Tesis propone la idea de ajustar dinámicamente la granularidad a la cual se determina el estado de la caché y se aplican las distintas políticas dependiendo de su comportamiento. El Adaptive Variable-Granularity Cooperative Caching (AVGCC) es el primer diseño capaz de adaptar la granularidad dependiendo del comportamiento de la caché para aplicar las políticas del ASCC. En un sistema con 4 procesadores, ejecutando cargas multiprogramadas, AVGCC consiguió una mejora en el rendimiento del 7.8 % con respecto al sistema base. Además, superó con creces el rendimiento de otros diseños suponiendo un coste de almacenamiento adicional mínimo; menos de un 0.2 %. También es importante destacar la reducción en la latencia media a memoria, 27 %, y en el consumo de potencia obtenido; menor en un 29 % al de una configuración tradicional. Cabe destacar también que se obtuvieron resultados similares utilizando cargas multihilo. Finalmente, esta Tesis demuestra que mejorar un diseño dotándolo de soporte para calidad de servicio, en inglés *Quality of Service*, no siempre implica una pérdida en el rendimiento medio. Un diseño extendido del AVGCC, denominado QoS-Aware AVGCC, obtuvo una mejora en el rendimiento de un 8.1 % a pesar de suponer un coste de almacenamiento adicional casi insignificante; 0.35 %.

Contribuciones

Las principales contribuciones de esta Tesis son:

1. Una nueva métrica sencilla y con un coste reducido, llamada *Set Saturation Level* o SSL, capaz de medir el grado en el que un conjunto de la caché es capaz de alojar su conjunto de trabajo particular y así detectar tanto desequilibrios entre conjuntos como problemas globales de capacidad.

2. Un novedoso diseño caché, llamado *Set Balancing Cache* o SBC, capaz de reducir fallos de conflicto mediante la creación de asociaciones entre conjuntos y de controlar el desplazamiento de líneas entre los mismos. Este diseño ha sido evaluado con éxito, superando a las técnicas más recientes en su campo.

Simulaciones exhaustivas han probado la capacidad de este diseño en distintos entornos además de su versatilidad.

3. Una combinación coordinada de diseños capaz de reducir fallos de conflicto y de capacidad al mismo tiempo. La *Bimodal Set Balancing Cache* o BSBC, la cual extiende las capacidades de la SBC para reducir fallos de conflicto incluyendo una política de inserción específicamente diseñada para reducir fallos de capacidad, ha sido implementada y evaluada exitosamente.

4. Una nueva técnica para equilibrar la cantidad de espacio destinada a instrucciones y datos en cachés de primer nivel: la *Virtually Split Cache* o VSC. Esta técnica combina la capacidad de compartición de recursos de aproximaciones unificadas con el alto ancho de banda y el paralelismo que proporciona una configuración separada.

5. Un nuevo entorno, la *Thread-Aware Bimodal Set Balancing Cache* o TABSBC, dotado de soporte para tener en cuenta el comportamiento de los distintos hilos que comparten recursos y que coordina de un modo razonado estrategias orientadas a reducir fallos de conflicto y de capacidad para cachés compartidas en procesadores multinúcleo.

6. Finalmente, esta Tesis introduce un nuevo esquema de cachés cooperativas para jerarquías de memoria privadas en CMP: el *Adaptive Set-Granular Cooperative Caching* o ASCC. Esta técnica combina un mecanismo de desplazamiento de líneas con la aplicación de una política de inserción específicamente diseñada para tratar con problemas de capacidad utilizando el SSL para controlar ambos comportamientos. Se propone también un nuevo estado para los conjuntos en el mecanismo de desplazamiento de líneas, el estado neutral, que impide que un conjunto pueda participar en los desplazamientos si esto puede resultar perjudicial. Además, se extiende esta técnica para dotarla de un mecanismo capaz de establecer la granularidad más adecuada a la cual se aplican las distintas políticas caché: el *Adaptive Variable-Granularity Cooperative Caching* o AVGCC.

Publicaciones derivadas de la Tesis

Revistas

- D. Rolán, B. B. Fraguela, and R. Doallo. Set Saturation Level-Based Cache Management. In *IEEE Transactions on Architecture and Code Optimization*, 2012 (En revisión).

- D. Rolán, B. B. Fraguela, and R. Doallo. Virtually Split Cache: An Efficient Mechanism to Distribute Instructions and Data in First-Level Caches. In *IEEE Transactions on Architecture and Code Optimization*, 2012 (En revisión).

Conferencias internacionales

- D. Rolán, B. B. Fraguela, and R. Doallo. Adaptive line placement with the Set Balancing Cache. In *Proceedings of the 42nd IEEE/ACM International Symposium on Microarchitecture* (MICRO 2009), pages 529–540, Diciembre 2009.

- D. Rolán, B. B. Fraguela, and R. Doallo. Reducing capacity and conflict misses using set saturation levels. In *Proceedings of the 17th International Conference on High Performance Computing* (HiPC 2010), Diciembre 2010.

- D. Rolán, B. B. Fraguela, and R. Doallo. Adaptive Set-Granular Coopera-

tive Caching. In *Proceedings of the 18th International Symposium on High Performance Computer Architecture* (HPCA 2012), pages 213–224, Febrero 2012.

Conferencias nacionales

- D. Rolán, B. B. Fraguela, and R. Doallo. Caché con Reequilibrio de Conjuntos. In *Actas de las XX Jornadas de Paralelismo*, pages 253–258, Septiembre 2009.

A Adolfo Rolán Barja, meu avó,
para o que sempre serei máis ca un "enxeñeiro".

Acknowledgments

It is embarrassing to admit that this Thesis [1] starts as lax as its cover page states. Although it lists my name as the single author, it is fair to say that the content and ideas in the following pages arose from the effort of many people, and to them, without exception, I owe my thanks.

My advisors gave me the opportunity and the confidence I needed to undertake this task and did their best to teach me how to get involved in research these past several years. Thanks to Ramón, also for guiding this work with all his experience, and Basilio, whose ability to see problems right through solutions is simply commendable.

Next and foremost, there is no word or sentence good enough to give thanks to my loving parents, who long before the University taught me how to fulfill any single wish by means of willpower, persistence and a bit of talent; and to the rest of my family, especially to my siblings. Their unflagging advocacy and optimism have sustained me all along these years.

Many thanks to all the people who belong to the close circle that surrounds the GAC group (specially the *Lab 0.2* guys and Guille, who has provided me with the LaTeX template for this Thesis, among other things). They have graciously shared their time and thoughts with me either in dinner parties, coffee breaks and trips.

I am also unspeakably thankful to those people who have warmly welcome me wherever I have been through these years, especially Jose Renau and the guys from the MASC group and their close friends (Ehsan, Elnaz, Amir, Rigo...) in California, and Marcelo and the CArD group staff back in Edinburgh.

I do not want to forget those people who enabled me to stop thinking about this work, even just for a while. Thanks to my school of languages classmates (especially

[1] This work was supported by the Xunta de Galicia under projects INCITE08PXIB105161PR and *"Consolidación e Estructuración de Unidades de Investigación Competitivas"* 3/2006 and 2010/06 and the MICINN, cofunded by the Fondo Social Europeo, under grants with references TIN2007-67536-C03-0 and TIN2010-16735. The author is also member of the HiPEAC network.

Héctor and Ali) and teachers, friends all, and to my outdoor and indoor soccer mates. Their support and friendship have been invaluable.

Last but not least, thanks to you, Raquel, for many times of steadfast friendship and affection. Let me humbly dedicate these ending lines to you in an attempt to begin another wonderful journey ;)

Dyer.

"Research is to see what everybody else has seen,
and to think what nobody else has thought"
Albert Szent-Gyorgyi

Abstract

Efficient memory hierarchy design is critical due to the large difference between the speed of the processors and the memory. In this context, cache memories play a crucial role bridging this gap. Cache management has become even more significant due to the appearance of chip multiprocessors (CMPs), which imply larger memory bandwidth requirements and greater working sets of many emerging applications, and which also need a fair and efficient distribution of cache resources between the cores in a single chip.

This dissertation aims to analyze some of the problems commonly found in modern caches and to propose cost-effective solutions to improve their performance. Most of the approaches proposed in this Thesis reduce cache miss rates by taking advantage of the different levels of demand cache sets may experience. This way, lines are placed in underutilized cache blocks of other cache sets if they are likely to be reused in the near future and there is no enough space in their native cache set. When this does not suffice, this dissertation proposes to modify in a coordinated way the insertion policies of oversubscribed sets. Hence, our proposals retain the most useful part of the working set in the cache while discarding temporary data as soon as possible. These ideas, initially developed in the context of last-level caches (LLCs) in single core systems, are successfully adapted in this Thesis to first-level caches and multicore systems. Regarding first-level caches, a novel design that allows to dynamically allocate banks to the instruction or the data cache depending on their degree of pressure is presented. As for multicore systems, our designs are firstly provided with thread-awareness in shared caches in order to give a particular treatment to each stream of requests depending on its owner. Finally, we explore the sharing of resources by means of the spilling of lines among private LLCs in CMPs using several innovative features such as a neutral state, which prevents caches from taking part in the spilling mechanism if this could be harmful, variable granularities

for the management of the caches, or the coordinated management of the cache insertion policy. Throughout this process we have used a simple and cost-effective metric to track the state of each cache set called Set Saturation Level (SSL). Finally, it is worthy to point out that our approaches are very competitive and often outperform many of the most recent techniques in the field, despite they imply really small storage and power consumption overheads.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

For many years, the memory hierarchy has been one of the main system performance bottlenecks. The lower the accesses are satisfied in this hierarchy, the more cycles and units of energy are consumed. Therefore, the efficiency of the memory hierarchy strongly depends on satisfying memory requests from the on-chip caches. This continues to be valid nowadays. In fact, the recent appearance of chip multiprocessors (CMPs) implies more pressure on the memory hierarchy and the number of cores that must be served is expected to increase rapidly, becoming thus even more important to avoid off-chip misses. Future last-level caches (LLCs) are expected to be larger and to occupy greater portions of the die area, which potentially means higher latencies and power consumption, depending on the distance traveled. Also, caches, and LLCs in particular, may be configured either as private to a thread or shared by multiple threads. As a result, forthcoming cache design strategies must undertake new challenges in order to adapt to multicore platforms and parallel workloads of many emerging applications. This way, cache policies should now take into account in their design the interaction among threads as well as larger and non-uniform latencies and energy for data transfers inside the system.

This chapter covers the different cache organizations and configurations up to the present along with cache memory basics, followed by a brief description of the most representative work related to the exploration of new cache management policies. The problem this Thesis deals with and the statement it proposes are discussed next. Finally, the main contributions of this dissertation and a brief outline of the

contents found in it conclude this chapter.

1.1. Cache Memories: Basics

The concept of CPU cache was firstly introduced in the mid sixties [80], as a way of bridging the gap between processor and main memory speeds. Cache memories can be organized as *direct-mapped*, *set-associative* or *fully-associative*, depending on how many places one line can be mapped to, and their operation is endorsed by the principle of locality of references. This principle advocates that recently referenced data or instructions are likely to be referenced again in the near future, *principle of temporal locality*, and the same happens with the items located in nearby memory locations, *principle of spatial locality*. Smith [66] defined the concept of cache memory evaluating its performance as a function of its basic parameters: *cache size*, *associativity* and *line size*. Hill [20] classified cache misses into the traditional 3C model: *compulsory* or *cold misses*, those corresponding to the number of cache lines in the working set, *conflict misses*, which appear when the cache organization is not fully-associative, and, finally, *capacity misses*, which occur when the number of lines in the working set is greater than the number of lines that the cache can accommodate. Cache resources are typically adjusted following the replacement policy. The replacement process involves the following three primary policies:

- **Victim Selection Policy**: This policy selects a victim line to be evicted in order to make room for the line requested in a cache miss.
- **Insertion Policy**: It decides which is the eviction priority within a set that is assigned to the new line inserted due to a cache miss.
- **Promotion Policy**: This policy decides how the eviction priority should be modified when a hit occurs.

Current caches usually apply the Least Recently Used (LRU) replacement policy, or some kind of variation [68]. This policy requires the maintenance of a recency stack for each set to indicate the ordering of the last access to each block in that set, while it inserts new lines and always promotes the last accessed line to the Most

Recently Used (MRU) position, which is the one with the highest priority in the cache, i.e., the less likely to be evicted.

Moreover, cache memories can accommodate instructions, data or both, in which case they are known as *unified* caches. Instructions are mainly read-only, while data can be read or written, thus, caches must support these two essential operations. Caches can apply two different write policies, namely *write-trough*, if every single write operation in the upper levels is propagated to the lower ones, or *write-back*, if this is done only when a dirty line is evicted. Another important feature of the cache memory hierarchy is the *inclusivity* of its levels. Those memory hierarchies that are *inclusive* force every lower level in the hierarchy to accommodate all lines in the upper ones, while *exclusive* ones are not that restrictive in order to increase the effective capacity of the total memory hierarchy. The inclusivity of each level impacts its main underlying policies, namely write and replacement policies, as well as the coherence protocol.

Cache memories can also be classified according to other characteristics. For example, depending on their access time they can belong to one of two main groups, namely *Uniform Cache Access* (UCA) or *Non-Uniform Cache Access* (NUCA). Until relatively recently, processors used to include only cache structures able to feed them with data in a fixed time regardless of the line being accessed. That is, the access latency of these caches was fixed to guarantee the longest possible delay for any line. This way, the issue logic is simplified, especially in the first level of the cache memory hierarchy since the processor schedules instructions based on its hit latency. Nevertheless, as caches become larger and are composed of many more banks, it is not sensible to force every cache access to incur the delay penalty of accessing the furthest bank in the cache. By applying some modifications to the traditional structures, such as allowing variable access times, cache memories can support non-uniform accesses [40]. Actually, many of the cache organizations that will be discussed further in this chapter are examples of NUCA architectures.

Also, CMPs have caches that can be configured as either shared among several cores or private to each one of them. Figure 1.1 shows both a private (a) and a shared (b) configuration for the L2 cache in a 4-core CMP. First-level caches are not usually shared, since every single core needs to access them almost in every cycle. Selecting either a private or a shared LLC is one of the open dilemmas in computer

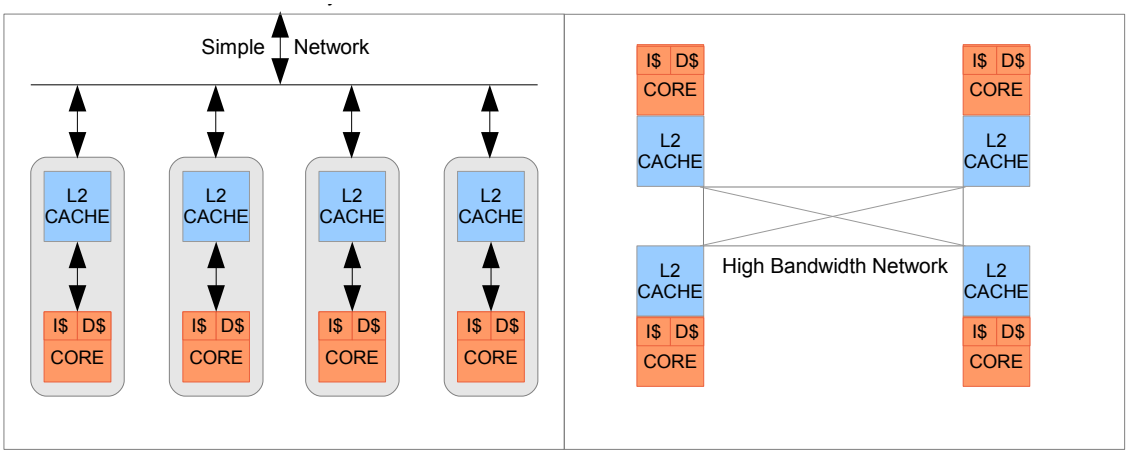


Figure 1.1: Shared vs. private cache configuration

architecture. Private configurations provide isolation among applications, simpler optimizations and lower average latencies than shared approaches, whereas they mean low effective capacity both locally to each core, since their size is fixed at design time, and globally to the whole system, due to data replication. On the other hand, shared configurations automatically distribute resources and provide higher effective capacity than private approaches, but they suffer due to the interaction between threads and need high average latencies. Also, private configurations rely on more complex coherence protocols while shared LLCs may require huge interconnection networks.

1.2. Related Work

At the inception of cache memories, Belady evaluated an ideal cache scheme [5] in order to minimize the number of cache misses. Although this proposal is not feasible nowadays, as it relies on future knowledge of the stream of references, the study showed that there was room for improvement over the traditional cache design. Since then, many proposals have appeared in order to increase cache performance in both single and multicore environments. This section describes some of this work.

The bibliography that is more strongly connected with the techniques proposed

in this Thesis, and the problems they deal with, will be reviewed in more detail in the corresponding chapters.

1.2.1. Related Work in Single Core Environments

Research efforts were first addressed to increase performance in first-level caches, which are usually small-sized, have low degrees of associativity and determine the cycle time of the processor. These works focused on adding expensive hardware structures, like prefetchers [14][32][50] to reduce compulsory misses, on experimenting with different cache organizations [2][6], on improving cache management [16][29][44] or on exploiting memory level parallelism [41]. As the cache size, associativity and number of levels in the memory hierarchy increased, efforts were aimed at reducing the number of cache misses by applying simpler cache policies, which usually sought to keep more accurately as much as possible of the working set in the cache. The idea underlying most proposals to improve the capability of the caches to keep the working set is to increase the number of possible places where a memory block can be placed with respect to the standard design, that is, increasing its effective capacity. The most representative related work that focuses on reducing conflict and capacity misses is described next.

1.2.1.1. Reducing Conflict Misses

The non-uniformity that memory references exhibit in general purpose applications creates an unbalanced demand across the cache sets, resulting in conflict misses. The impossibility for some cache sets to hold their working set has been addressed by victim caches [34], which simply store the latest lines evicted from the cache. This idea has been later refined with heuristics to decide which lines to store in the victim cache. For example, [22] takes its decisions based on reload intervals, while [3] considers the frequency with which each line appears in the miss stream.

Alternative indexing functions [38][65] that succeed at achieving a more uniform distribution of the references across the cache have been suggested.

In general, the smaller the associativity of the cache, the greater the imbalance in the demand on the individual sets of the cache. Thus it was in the context of direct-

mapped caches where the first approaches of this kind appeared. Pseudo-associative caches belong to this family of proposals. Initially they provided the possibility of placing the blocks in a second associated cache line, providing a performance similar to that of 2-way caches [2][6], but they were also generalized to provide larger associativities [87].

The adaptive group-associative cache (AGAC) [51] tries to detect underutilized cache frames in direct-mapped caches in order to retain in them some of the lines that are to be replaced. AGAC records the location of each line that has been displaced from its direct-mapped position in a table, which is accessed in parallel with the tag-storage.

The Indirect Index Cache (IIC) [18] seeks maximum flexibility in the placement of memory blocks in the cache. Its tag-store entries keep pointers so that any tag-entry can be associated to any data-entry.

The NuRAPID cache [10] provides a flexible placement of the data-entries in the data array in order to reduce average access latency, allowing the most recently used lines to be in the fastest subarrays in the cache.

The B-Cache [86] tries to reduce conflict misses balancing the accesses to the sets of first-level direct-mapped caches by increasing the decoder length and incorporating programmable decoders and a replacement policy to the design.

The V-Way cache [55] adapts to the non-uniform distribution of the accesses on the cache sets by allowing different cache sets to have a different number of lines according to their demand.

More recently, Scavenger [3] has been proposed, which is exclusively oriented to last-level caches and partitions the cache in two halves. One half is a standard cache, while the other half is a large victim file organized as a direct-mapped hash table with chaining, in order to provide full associativity. This approach also tackles capacity misses to some extent, as it learns to retain the most frequently missing blocks.

The Z-Cache [64] decouples the notion of ways and associativity. Ways represent the number of tags that must be searched when looking for a cache line, while associativity is referred as the number of blocks that could be evicted to make room

for an incoming line. The Z-Cache keeps the number of ways small, but it has a large associativity. It is based on a skewed-associative cache design [65], where each way of the cache uses a different indexing (hashing) function.

1.2.1.2. Reducing Capacity Misses

The proposals we have just discussed emphasize the flexibility of placement of lines in the cache to improve miss rates or access time. Other researchers have focused on the modification of the replacement policy in order to keep the most useful lines in the set where they belong. For example, several adaptive insertion policies were proposed in [53], the Dynamic Insertion Policy (DIP) being the greatest exponent. This technique uses set dueling, which devotes a few cache sets to track the behavior of two insertion policies, namely the traditional MRU one and the Bimodal Insertion Policy (BIP), and dynamically chooses the best one for the rest of the sets. The BIP policy inserts the new lines most of the times in the LRU position and only with a low probability in the MRU one in order to discard temporary data and retain the most useful part of the working set in the cache. Other techniques change the replacement policy taking into account not only recency, as the traditional LRU policy does, but also frequency [45] or considering the impact in performance of a certain cost parameter [29][72], like the latency needed to access remote or local blocks in a NUCA architecture.

Pseudo-LIFO replacement policies [9] evict blocks from the upper part of a fill stack, i.e., among the most recently inserted lines of the set. This contributes to retain a large fraction of the working set in the cache.

More recently, DRRIP [28] chooses the most appropriate Re-Reference Interval Prediction (RRIP) insertion policy for each application based on set dueling. It uses multiple bits per line to track priorities at a fine granularity. A low value implies that the line is expected to be re-referenced in the near future, and a high value means that the line is likely to be accessed in the distant future.

In STEM [85] each set in the tag array is augmented with partial (hashed) tags of recently evicted blocks. Saturating counters keep track of hits in these victim tags and determine if the set should spill or receive blocks.

Furthermore, techniques which improve cache management by bypassing lines which are not likely to be referenced if they are installed in the cache [16][30][31][39][77] or by evicting earlier those lines that are predicted to be dead or that have less locality [35][39][81], have successfully proved to reduce capacity misses.

1.2.2. Related Work in Multicore Platforms

The increasing number of cores per chip has led to huge power consumptions, which has translated into processors clocked at lower frequencies in order to save energy. Despite that, the difference between memory and processor speeds has not disappeared. This fact, along with the higher pressure that multicore processors place on the memory system, makes it even more important to explore new cache design strategies to minimize accesses to the lower levels of the memory hierarchy. This way, novel approaches, many of which are not suitable for the traditional multiprocessors, have been proposed to exploit the new capabilities of CMPs, focused on the memory hierarchy. Many of them have adapted techniques originally designed to work in single core environments to the shared and private levels in the cache memory hierarchy of multicore systems. By using private Last-Level Caches (LLCs), a system is able to provide the applications with isolation, low latency and minimum bandwidth while easier design extensions are also allowed. Shared configurations provide the system with the ability to share resources, adapting dynamically their allocation depending on the demand of each core. It is also worthy to emphasize that many processors may employ combinations of private and shared caches in different levels of the memory hierarchy.

1.2.2.1. Cache Memory Hierarchies with Shared Levels

There have been many software [17][25] and hardware [8][12][54] proposals to optimize the behavior of shared caches by partitioning their resources among the applications that share them. Moreover, Zhang and Asanovic [88] propose a simple mechanism to implement block replication without incurring much overhead for coherence among replicas in the shared cache.

Adaptive Set Pinning (ASP) [70] reduces misses thanks to the ownership of each

cache set by a processor, which is the only one that can insert new lines in it, the other processors having to resort to a small processor owned private (POP) cache partition.

Other approaches focus on adapting policies designed for private caches in single core environments to shared caches. The Thread-Aware Dynamic Insertion Policy (TADIP) [27] extends to shared caches the insertion policies introduced in [53] to deal with capacity misses in private caches. TADIP can apply these policies in isolation to each independent thread according to the benefit it can get from them, measured by means of set dueling. The thread-aware TA-DRRIP [28] also chooses the most appropriate Re-Reference Interval Prediction (RRIP) insertion policy for each application based on set dueling. The Pseudo-LIFO policies [9] have also been successfully evaluated in shared caches.

Furthermore, there are approaches like Promotion/Insertion Pseudo-Partitioning (PIPP) [84], which combines pseudo-partitioning with new insertion and promotion policies.

1.2.2.2. Cache Memory Hierarchies with Private Levels

Many of the approaches that have appeared in the past years for CMPs focused on providing private levels with shared capacity. Some techniques use partitioning in order to limit the amount of space for private and shared data while others try to make a better usage of resources by spilling lines between caches, even existing mixed approaches like the Elastic Cooperative Caching (ECC) [19]. ECC splits sets in two different regions, a private one, to allocate lines evicted from the upper level, and a shared region, to hold lines spilled by neighbor caches.

Regarding the designs that only rely on spills, Cooperative Caching (CC) [7] spills lines to other caches instead of directly evicting them to main memory if they are the only copy in the chip. Dynamic Spill-Receive [52] (DSR) labels each cache as either spiller or receiver depending on a global counter per cache used by its set dueling mechanism. This global counter is updated by all the caches in order to determine whether the spillings are going to hurt receiver caches or not.

The Adaptive Placement Policy (APP) [63] scheme learns from the past cache

behavior to make a better decision on whether to place a newly fetched block in order to convert many of the remote cache hits into local ones.

As for the partitioning-oriented approaches, Adaptive Selective Replication [4] dynamically analyzes the workload behavior and adapts the degree of replication on a per block basis to match the application requirements. In Cooperative Cache Partitioning [8] resources are partitioned both in terms of time, giving different priorities of execution to different partitions, and space, setting the size of the different partitions depending on the application requirements.

Recently, the MorphCache [71] allows reconfiguration to form larger caches. Depending on an analysis of the working set requirements, slices of the lower levels of the cache memory hierarchy are either merged to form a larger shared cache slice or previously merged slices are split into smaller private slices. Thus, each cache slice may either be a private cache or it may be a part of a larger cache. Merging is triggered when there is unbalance in the usage of adjacent slices or if highly-utilized slices are dealing with the same shared data.

1.3. The Problem

Cache memories play an essential role in the performance of single core systems by bridging the gap between processor speed and main memory latency. Modern processors include several cache levels with larger sizes and latencies the lower they are located in the hierarchy. Their role is to retain the current working set closer to the processor with different access costs depending on the level of locality of the data. First-level caches are strongly restricted by their access time requirements but current processors are able to hide most of their latency using out-of order execution as well as miss overlapping techniques. On the other hand, the last levels of the cache memory hierarchy are not so restricted by their access time, as the processor does not schedule instructions based on this. Rather, their design focuses on retaining the working set of the applications, the locality in these lower levels being filtered by the upper levels. The main reason behind the emphasis on the adaptation to the locality of the applications in these levels is that as requests travel down in the memory hierarchy they require a greater number of cycles to be satisfied,

so it becomes more difficult to hide the latency of last-level caches. In multicore systems the importance of caches is even larger due to the growing number of cores that share the bandwidth that the main memory can provide. In an attempt to make a more efficient usage of their caches, the memory hierarchies of many chip multiprocessors have LLCs shared by several cores. As a result, these LLCs can hold both data private to each thread running in each core, as well as data shared among several threads. Despite these emerging configurations, modern designs keep using traditional policies to manage large and shared last-level caches, which often leads to suboptimal behaviors that degrade the overall performance. For this reason, the improvement of such designs in order to avoid their inefficiencies and to better adapt them to these emerging architectures is an active area of research.

One of the most commonly observed behaviors is that memory references are often not uniformly distributed across the sets of a set-associative cache, the most common design nowadays [55]. As a result, at a given point during the execution of a program there are usually sets whose working set is larger than their number of lines (the associativity of the cache), while the situation in other sets is exactly the opposite. The outcome of this is that some sets exhibit large local miss ratios because they do not have the number of lines they need [26], while other sets achieve good local miss ratios at the expense of a poor usage of their lines, because some or many of them are actually not needed to keep the working set. Cache performance can be substantially improved by associating these two kinds of sets in order to balance their working sets. This dissertation poses a new cache design, called *Set Balancing Cache* (SBC), aimed to reduce the number of conflict misses by displacing lines from oversubscribed sets to underutilized ones.

In order to perform associations, the level to which a set is able to hold its particular working set must be measured. The hardware overhead for tracking the behavior of all sets using extra tags can be prohibitively expensive. Large and shared caches typically have thousands of sets. This Thesis proposes a metric, named *Set Saturation Level* (SSL), based on saturation counters instead. This dissertation shows how this metric is able to track the behavior of a given set by leveraging the principles of locality while it enables cost-effective cache optimizations.

Another problem commonly found in cache memories is thrashing, which occurs when a stream of temporary references evicts lines with more locality from the cache,

keeping useless data and increasing the number of capacity misses. Some proposals have appeared in the last years specifically oriented to reduce capacity misses. A very good example is [53], which targets memory-intensive workloads with working sets that do not fit in the cache for which the traditional LRU replacement policy is counterproductive. It introduces a new insertion policy specifically designed to deal with thrashing. In this dissertation the *Bimodal Set Balancing Cache* (BSBC), a coordinated strategy to reduce both capacity and conflict misses by changing the placement and insertion policies of the cache, is presented. This strategy uses the Set Saturation Level to control both policies and their operation. As the SSL tracks the behavior of an individual set in terms of its ability to hold its particular working set, considering all SSLs in the cache provides an idea of the capacity the whole cache is able to endure. This Thesis evaluates the goodness of the SSL not only as indicator of unbalances between sets but also as a measure to detect global problems of capacity in the cache. It also shows that the combination of several techniques, which were devised to deal with different problems, without coordination can be harmful for the overall performance. Furthermore, experiments in multicore environments highlight the importance of treating different access streams, like those generated by several threads sharing resources, in a different way in response to their specific behaviors.

The idea of applying different policies or limiting the allocation of resources to the different access streams that a cache may experience is adaptable to first-level traditional caches, where the two main streams are those due to instructions and data. In this context, a key point of design commonly accepted nowadays is to split first-level caches for both instructions and data instead of unifying them in a single cache. Although that approach eases the pipeline design and requires less complexity than a unified approach it also reduces the global hit rate. This Thesis proposes a new technique, called *Virtually Split Cache* (VSC), which decides the amount of space devoted to instructions and data with the purpose of optimizing performance in first-level set-associative caches by dynamically adjusting the allocation of resources depending on their particular demand.

The two previously mentioned sources of inefficiency, the non-uniform distribution of the memory accesses across the cache sets and thrashing, are also found in the shared caches of CMPs. These caches also have different access streams due to several threads that share the cache resources. This dissertation shows that standard

management strategies for private caches, which are by nature thread-oblivious, often lead to suboptimal behaviors when applied to shared caches. A new design for shared LLCs in multicore processors, called *Thread-Aware Bimodal Set Balancing Cache* (TABSBC), based in the SBC insights to deal with conflict misses, with a thread-aware mechanism as well as an insertion policy specifically oriented to reduce the effects of thrashing, is also presented and evaluated.

Later, this dissertation provides a framework that adapts the knowledge gathered in the previous steps of the Thesis to a common CMP with a private cache memory hierarchy for each core. Choosing either a private or a shared configuration for the last-level cache (LLC) is one of the key points of the design in CMPs. When the LLC is shared among all the cores, it requires high bandwidth because every single request by any upper cache needs to access the interconnection network. Shared LLCs are usually distributed in tiles owned by different cores and, thus, needing different latencies depending on where the requested line is found. As the number of cores and cache banks increases, it becomes more difficult to hide wire delays. Even worse, harmful applications can hurt the performance of other concurrently executing applications. On the other hand, in private configurations, each core is assigned a static portion of the LLC, which provides lower latency, better scalability, isolation and makes the optimization of particular parameters, like power consumption, easier, at the cost of depriving the system of the ability of sharing underutilized resources. CMPs provide room for improving performance by managing the allocation of resources to the multiple different applications that can be executed concurrently, as some of them can be short of cache resources while others can offer underutilized space. Therefore, it is interesting to track the global availability of resources and select the best policies to allocate them appropriately.

Several proposals have been presented in order to share resources in private configurations by displacing or spilling lines from one cache to another [7][52]. This Thesis proposes *Adaptive Set-Granular Cooperative Caching* (ASCC), which measures the degree of stress of each set and performs spills between spiller and potential receiver sets, while it tackles capacity problems as well. Also, it adds a neutral state to prevent sets from being either spillers or receivers when it could be harmful.

Furthermore, for some workloads, decisions work better when they are taken globally, after tracking the state of the whole cache and applying policies uniformly,

while in other situations finer granularities, for example at a set level, provide the best results. This Thesis shows the importance of applying different granularities and introduces the *Adaptive Variable-Granularity Cooperative Caching* (AVGCC), which dynamically adjusts the granularity for applying the ASCC policies.

Finally, it is worthy to emphasize that every single approach proposed in this Thesis implies simple and feasible changes in the traditional cache memory operation as well as negligible storage and power consumption overheads.

1.4. Thesis Statement

There is room for improvement in traditional cache designs in order to better adapt to the particular characteristics of the different access streams that cache memories may experience in both single and multicore environments, either for private or shared configurations. Simple and cost-effective changes to cache management can substantially improve their performance.

1.5. Contributions

The main contributions of this Thesis are:

1. A simple and cost-effective metric, the *Set Saturation Level* or SSL, which measures the degree to which a set is able to hold its particular working set, has been proposed. It proved to be successful at detecting unbalances between sets as well as global capacity problems.

2. A novel structure for cache memories aimed to reduce cache misses by managing associations among sets and controlling the displacement of lines between them has been designed: the *Set Balancing Cache* or SBC. This design was successfully evaluated, outperforming the most recent designs in the field. Simulation results have shown the goodness of our approach and its versatility in the different environments tested.

3. This Thesis shows that cache performance can be notably improved by tackling conflict and capacity misses at the same time. The *Bimodal Set Balancing Cache* or BSBC, which extends the SBC capabilities to reduce conflict misses with an insertion policy specifically designed to deal with capacity misses, has been successfully evaluated.

4. A new technique aimed to balance the amount of space devoted to instructions and data for optimizing performance in first level set-associative caches: the *Virtually Split Cache* or VSC. This technique combines the sharing of resources typical of unified approaches with the high bandwidth and parallelism that split configurations provide.

5. A new framework, the *Thread-Aware Bimodal Set Balancing Cache* or TAB-SBC, that coordinates in a sensible way strategies to reduce conflict and capacity misses by means of a thread-aware mechanism designed for the shared caches of chip multiprocessors (CMPs).

6. Finally, this Thesis introduces a new cooperative caching alternative in CMPs with a private cache memory hierarchy for each core: the *Adaptive Set-Granular Cooperative Caching* or ASCC. This design combines a spilling mechanism with and insertion policy specifically designed to tackle capacity problems, both policies being controlled by the SSL metric. A neutral state to prevent sets from taking part in the spilling mechanism when it may not be beneficial is also proposed. Furthermore, an extended design, the *Adaptive Variable-Granularity Cooperative Caching* or AVGCC, able to use the most suitable granularity to track the state of the cache and apply the best policies is proposed.

1.6. Overview of the Contents

The Thesis is organized into seven chapters, this one included, whose contents are summarized next.

Each one of the following five chapters is organized as follows: they include, firstly, a brief introduction section to detail the background and motivation of each

approach, secondly, the description of each implementation and its corresponding evaluation and, finally, a summary with the main ideas and results obtained.

Chapter 2 motivates and introduces the basics of the *Set Balancing Cache* (SBC) as well as the *Set Saturation Level* (SSL) metric. In this chapter the SBC performance and the SSL goodness are thoroughly evaluated in a single core environment.

Chapter 3 extends the SBC with support against thrashing by providing it with a suitable insertion policy specifically designed to reduce capacity misses. This design, called *Bimodal Set Balancing Cache* (BSBC), is evaluated in both private and shared caches, concluding that it performs well on both although it is outperformed in shared caches by techniques equipped with thread-aware mechanisms.

Chapter 4 adapts the concept of recognizing the different access streams generated by simultaneous running threads in shared caches to first-level traditional ones, where the two main streams are due to instructions and data. This chapter introduces the *Virtually Split Cache* (VSC), which is able to dynamically adjust cache resources devoted to instructions and data depending on their particular demand.

Chapter 5 provides a coordinated mechanism with thread-aware support for shared caches without implying performance losses: the Thread-Aware Bimodal Set Balancing Cache (TABSBC).

Chapter 6 discusses a new cooperative caching implementation for chip multiprocessors with private last-level caches, the Adaptive *Set-Granular Cooperative Caching* (ASCC), and an extended design able to apply the best granularity with which the cache state should be tracked depending on the running application, the *Adaptive Variable-Granularity Cooperative Caching* (AVGCC).

Finally, Chapter 7, is devoted to the main conclusions of the Thesis and directions for future work.

Chapter 2

Set Balancing Cache

2.1. Introduction

Memory references are often not uniformly distributed across the sets of a set-associative cache, the most common design nowadays [55]. As a result, at a given point during the execution of a program there are usually sets whose working set is larger than their number of lines (the associativity of the cache), while the situation in other sets is exactly the opposite. The outcome of this is that some sets exhibit large local miss ratios because they do not have the number of lines they need [26], while other sets achieve relative good local miss ratios at the expense of a poor usage of their lines, because some or many of them are actually not needed to keep the working set. An intuitive answer to this problem is to increase the associativity of the cache. Multiplying by n the associativity is equivalent to merging n sets in a single one, joining not only all their lines, but also their corresponding working sets. This allows to balance smaller working sets with larger ones, making available previous underutilized lines for the latter, which results in smaller miss rates. Unfortunately, increments in associativity impact negatively access latency and power consumption (e.g. more tags have to be read and compared in each access) as well as cache area, besides increasing the cost and complexity of the replacement algorithm. Worse, progressive increments in the associativity provide diminishing returns in miss rate reduction, as in general, the larger (and fewer) the sets are, the more similar or balanced their working sets tend to be. This way, only

restricted levels of associativity are found in current caches.

This chapter proposes an approach to associate cache sets whose working set does not seem to fit in them with sets whose working set fits, enabling the former to make use of the underutilized lines of the latter. Namely, this cache design, called Set Balancing Cache or SBC, shifts lines from sets with high local miss rates to sets with underutilized lines where they can be found later. This process is done by setting the role of sets in the associations relying in a new cost-effective metric called Set Saturation Level or SSL, which measures the degree to which a set is able to hold its working set by means of a saturation counter per set. Notice that while an increase in associativity equates to merging sets in an indiscriminate way, our approach only exploits jointly the resources of several sets when it seems to be beneficial. Also, increases in associativity cannot choose which sets to merge, while the SBC can be implemented using either a static policy, which also preestablishes which sets can be associated, or a dynamic one that allows to associate a set with any other one. Thus, as we will see in the evaluation, the SBC achieves better performance than equivalent increases in associativity while not bringing their inconveniences.

2.2. Background and Motivation

There have been several proposals to improve the architecture of caches to deal with the problem of the non-uniform distribution of memory accesses across the cache sets. For example, alternative indexing functions have been suggested [38][64][65], but they do not attempt to identify underutilized lines or working sets that cannot be retained successfully in the cache. The idea underlying most proposals to improve the capability of the caches to keep the working set is the increase, with respect to the standard design, of the number of available places where a memory block can be placed. While most of them have been already introduced in Section 1.2.1.1, we will discuss here the most related ones to our SBC, which will be further described in the next section.

Pseudo-associative caches [2][6][87] perform searches line by line. This way, they have search structures at the line level. Besides they do not provide mechanisms to inhibit line displacements: whenever a cache line is occupied by a memory block

mapped to it and a second memory block of this kind is requested, there is an automatic displacement to an associated cache line. Finally, all pseudo-associative caches swap cache lines under non-first hits in order to place them back in their major location according to the default mapping algorithm of the cache, so that successive searches will find them in the first search.

The V-Way cache [55] duplicates the number of sets and tag-store entries, keeping the same associativity and number of data lines. Data lines are assigned dynamically to sets depending on the access pattern of the sets and a global replacement algorithm on the data lines. Namely, the V-Way cache reassigns the less reused data lines to sets with empty tag-store entries that suffer a miss, which is the origin of the variability of the set sizes. When a set reaches its maximum size, it stops growing and replacements take place under a typical replacement algorithm such as LRU. The structure to allow any data line to be assigned to any tag-entry requires the storage for forward and reverse pointers between the tag-store and the data-store entries, besides the reuse counters used by the global replacement algorithm.

Scavenger [3] is exclusively oriented to last-level caches and partitions the cache in two halves. One half is a standard cache, while the other half is a large victim file (VF). The VF tries to retain the blocks that miss more often in the conventional cache, which are identified by a skewed bloom filter based on the frequency of appearance of each block in the sequence of misses. If a block evicted from the standard cache is predicted by the filter to have more misses than the block with the smaller priority in the VF, this latter block is replaced by the one evicted from the standard cache. This policy requires a priority queue that maintains the priorities of all the VF blocks. Accesses take place in parallel in both halves of the cache. When a block is found in the VF, it is moved to the standard cache.

As we see, there is no previous work on tracking the individual state of each cache set in order to detect unbalances in their demand with cost-effective metrics and, with this information, performing displacements of lines between complementary sets by setting associations when this could be beneficial.

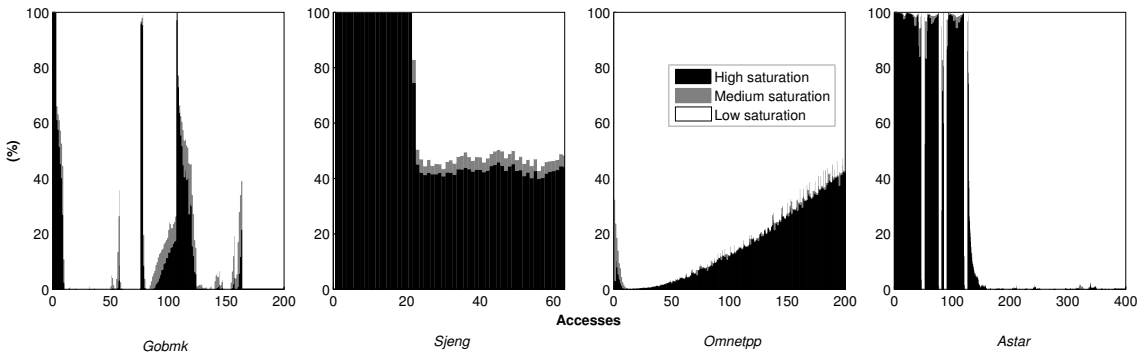


Figure 2.1: Distribution of the sets with a high saturation level(black), medium saturation level(gray) and low saturation level(white) in *445.gobmk*, *458.sjeng*, *471.omnetpp* and *473.astar*. Samples each $5 * 10^5$K accesses.

2.3. Static Set Balancing Cache

We seek to reduce the pressure on the cache sets that are unable to hold all the lines in their working set, by displacing some of those lines to sets that seem to have underutilized lines. These latter sets are those whose working set fits well in them, giving place to small local miss rates. This idea requires in the first place a mechanism to measure the degree to which a cache set is able to hold its working set. This is the saturation level of the set and it is measured by means of a counter with saturating arithmetic that is modified each time the set is accessed. If the access results in a miss, the counter is increased, otherwise it is decreased. We will refer to this counter as saturation counter.

The fact that different sets can experience very different levels of demand has already been discussed in the bibliography [51][55]. This fact, which is the base for our proposal, can be illustrated with the saturation counters. Figure 2.1 classifies the sets in a 8-way 2MB cache with lines of 64 bytes during the execution of four benchmarks from the SPEC CPU2006 suite. The classification is a function of their saturation level as measured by saturation counters whose maximum value is 15 in this case. The levels of saturation considered are low (the counter is between 0 and 5), medium (between 6 and 10) and high (between 11 and 15). We can see how there are some sets that are little saturated, while others are very saturated, and both percentages vary over time. These sets of opposite kinds could be associated,

moving lines from highly saturated sets to little saturated ones in order to balance their saturation level and avoid misses.

Our approach is based on the idea of performing associations between cache sets with complementary saturation levels in order to make the most of cache resources. This also gives place to make second searches, or in general up to n-th searches if n sets are associated, whenever a line is not found in the set indicated by the cache indexing function and this set is known to have shifted lines to other set(s). As a result, the operation of the Set Balancing Cache involves, besides the saturation counters explained, an *association algorithm*, which decides which set(s) are to be associated in the displacements, a *displacement algorithm* which decides when to displace lines to an associated set, and finally, modifications to the standard cache *search algorithm*. We now explain them in turn.

2.3.1. Association algorithm

This algorithm determines to which sets can displace lines a given one. Although the number of sets involved could be any, and it could change over time, we have started studying the simplest approach, in which each cache set is statically associated to another specific set in the cache. That is the reason why this first design of our proposal is called static SBC (SSBC). This design minimizes the additional hardware involved as well as the changes required in the search algorithm of the cache. We have decided the associated set to be the farthest set of the considered one in the cache, that is, the one whose index is obtained complementing the most significant bit of the index of the considered set. This decision is justified by the principle of spatial locality, as if a given set is highly saturated, it is probable its neighbors are in a similar situation. A consequence of this decision is that given two sets X and Y associated by this algorithm, sometimes lines will be displaced from X to Y, and vice versa, depending on the state of their saturation counters. Notice also that when the associativity of a cache design is multiplied by 2, this is equivalent to merging in a single set the same two sets that our policy associates, i.e., those that differ in the most significant bit of the index.

2.3.2. Displacement algorithm

A first issue to decide is when to perform displacements. In order to minimize the changes in the operation of the cache and take advantage of line evictions that take place in a natural way in the sets, we have chosen to perform the displacements when a line is evicted from a highly saturated set. Since the replacement algorithm we consider for the cache sets is LRU, as it is the most extended one, this means that the LRU line will not be sent to the lower level of the memory hierarchy; rather it will be actually displaced to another set.

It is intuitive that displacements should take place from sets with a high saturation level to little saturated sets. Three parameters must be selected for this policy: a concrete range for the saturation counter, from which value of the counter we consider that displacements should take place, and under which value we consider a set to be little saturated, and therefore a good candidate to receive displaced lines. It has been experimentally observed that a good upper limit for a saturation counter in a cache with associativity K is $2K - 1$, thus the saturation counters used in this Thesis work in the range 0 to $2K - 1$.

Regarding the triggering of the displacement of lines from a set, when its saturation counter has a value under its maximum it means that there have been hits in the set recently, thus it is possible its working set fits in it. Only when the counter adopts its maximum value will have most recent accesses (and particularly the most recent one) resulted in misses and it is safer to presume that the set is under pressure. Thus our SBC only tries to displace lines from sets whose saturation counter adopts its maximum value, which is another decision taken based on our experiments.

Finally, although it is the association algorithm's responsibility to choose which is the set that receives the lines in a displacement, it is clear that displacing lines to such set if/when its saturation counter is high can be counterproductive, since that indicates the lack of underutilized lines. In fact we could end up saturating a set that was working fine when trying to solve the problem of excess of load on another set. Thus a second condition required to perform a displacement is that the saturation counter of the receiver is below a given limit called *displacement limit*. It has been experimentally determined that the associativity K of the cache is a good displacement limit for the counters in the range 0 to $2K - 1$ used. Notice that

since displacements only take place as the result of line evictions, the access to the associated set saturation counter needed to verify this second condition can be made during the resolution of the miss that generates the eviction.

Regarding the local replacement algorithm of the set that receives the displaced line, the line is inserted as the most recently used one (MRU). The rationale is that since the displaced line comes from a stressed working set, while the working set of the destination set fits well in it, this line needs more priority than the lines already residing in the set. Besides this way n successive displacements from a set to another one insert n different lines in the destination set. If the displaced line were inserted as the least recently used one (LRU), each new displacement would evict the line inserted in the previous one if there were no intermediate hits in the displaced lines. Another advantage of this decision is that the sets apply the traditional insertion policy for all the insertions, thereby minimizing the changes in the design of the cache. It has been experimentally checked that the insertion in the MRU position yields better results than in the LRU one.

2.3.3. Search algorithm

In the SBC a set may hold both memory lines that correspond to it according to the standard mapping mechanism of the cache and lines that have been displaced from its associated set. Thus the unambiguous identification of a line in a set requires not only its tag, but also an additional *displaced bit* or d for short. This bit marks whether the line is native to the set, when it is 0, or it has been displaced from another set, when it is 1. Searches always begin examining the set associated by default to the line, testing for tag equality and $d = 0$. If the line is not found there, a second search is performed in the associated set, this time seeking tag equality and $d = 1$. If the second search is successful, a secondary hit is obtained.

Our proposal avoids unnecessary second searches by means of an additional *second search* (*sc*) bit per set that indicates whether its associated set may hold displaced lines. This bit is set when a displacement takes place. Its deactivation takes place when the associated set evicts a line, if the OR of its d bits changes from 1 to 0 as result of the eviction. Checking this condition and resetting the second search bit of the associated set is done in parallel with the resolution of the miss that gen-

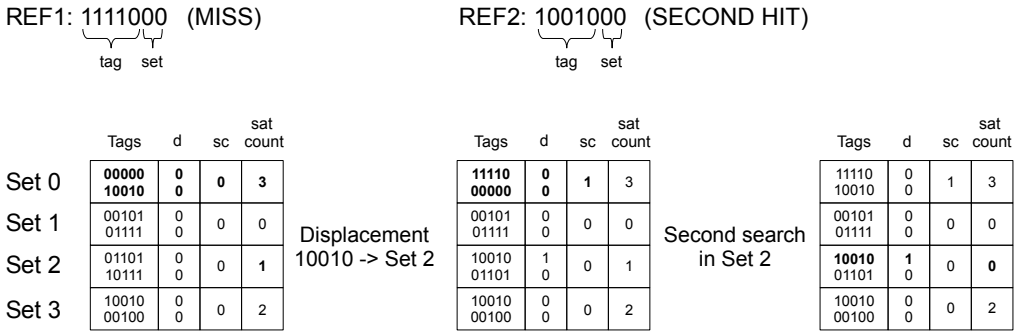


Figure 2.2: Static SBC operation in a 2-way cache with 4 sets. The upper tag in each set is the most recently used and the saturation counters operate in the range 0 to 3.

erates the eviction. Without this strategy to avoid unnecessary second searches, the IPC for the static SBC (SSBC) would have been 0.6% and 1.0% smaller in the two-level and the three-level cache configurations used in our evaluation in Section 2.6, respectively.

Contrary to other cache designs that lead to sequential searches in the cache [2][6] the SBC does not swap lines to return them to their original set when they are found displaced in another set. This simplifies management and does not hurt performance because our proposal, contrary to those ones, is oriented to non first-level caches. Thus once a hit is obtained in a line, the line is moved to the upper level of the memory hierarchy, where successive accesses can find it. Experiments performing swapping of lines in the SBC to return displaced lines to their original set under a hit proved that this policy had a negligible impact on performance.

2.3.4. Discussion

Figure 2.2 shows a simple example of the operation of a Set Balancing Cache in a 2-way cache with 4 sets. The upper tag in each set is the one of the most recently used line, and the saturation counters operate in the range 0 to 3. Line addresses of 7 bits are used for simplicity, the lower two bits being the set index and the upper 5 ones the tag. The first reference is mapped to set 0, where $sc = 0$, thus no second search is needed and a miss occurs. Checking saturation counters results

in a displacement of the line that must be evicted from set 0, here the one with tag 10010, to set 2 ($\overline{0}0 = 10$), so it is actually the LRU line of set 2 the one that is evicted from the cache. The second reference is mapped again to set 0, where it misses. Since now its *sc* bit is 1, a second search is performed in set 2, where there is a hit, since the tag is found with the displaced bit $d = 1$.

As Section 2.3.1 explains, a K-way SSBC associates exactly each pair of sets of the cache that would have been merged in a single set in the $2K$-way cache with the same size and line size. Still, there are very important differences between both caches. While the $2K$-way cache unconditionally merges the sets and their working sizes, in the SSBC the merging is conditioned by the behavior of the sets. Namely, their resources are shared only when at least one of the sets suffers a stream of accesses with so many misses that its saturation counter reaches the maximum limit, while the other set shows to be large enough to hold its current working set, which is signaled by a value of its saturation counter smaller than the displacement limit. This smarter management of the sharing of resources in the cache leads to better performance for the SSBC even when it leads to second accesses when lines have been displaced from their original sets.

Finally, in principle, the tag and data arrays of an SBC can be accessed in parallel. Still, we recommend and simulate a sequential access to these arrays for two reasons. One is that the SBC is oriented to non-first level caches, where both arrays are often accessed sequentially because in those caches the tag-array latency is much shorter than the data-array one, and the sequential access is much more energy-efficient than the parallel one [11][79]. The other is that since the SBC may lead to second searches, the corresponding parallel data-array accesses would further increase the waste of energy.

2.4. Dynamic Set Balancing Cache

The SSBC is very restrictive on associations. Each set only relies on another prefixed set as potential partner to help keep its working set in the cache. It could well happen that both sets were highly saturated while others are underutilized. When a cache set is very saturated, it would be better to have the freedom to

associate it to the more underutilized (i.e. with the smallest saturation value) non-associated set in the cache. This is what the dynamic SBC (DSBC) proposes. We now explain in turn the algorithms of this cache.

2.4.1. Association algorithm

The DSBC triggers the association of sets when the saturation counter of a set that is not associated with another set reaches its maximum value, which is $2K - 1$ in our experiments, where K is the associativity of the cache. When this happens, the DSBC tries to associate it with the available set (i.e. not yet associated with another one) with the smallest saturation level. An additional restriction is that the association will only take place if this smallest saturation level found is smaller than the displacement limit, described in Section 2.3.2. The reason is that it makes no sense to consider as candidate for association a set whose saturation counter indicates that lines from other sets should not be displaced to it.

In principle this policy would require hardware to compare the saturation counters of all the available sets in order to identify the smallest one. Instead we propose a much simpler and cheaper design that yields almost the same results, which we call *Destination Set Selector* (DSS). The DSS has a small table that tries to keep the data related to the least saturated cache sets. Each entry consists of a valid bit, which indicates whether the entry is valid, the index of the set the entry is associated to, and the saturation level of the set. Comparers combined with multiplexers in a tree structure allow to keep updated a register *min* with the minimum saturation level stored in the DSS (*min.level*), as well as the number of its DSS entry (*min.entry*) and the index of the associated set (*min.index*). This register provides the index of the best set available for an association when requested. Similarly, a register *max* with the maximum saturation counter in the DSS (*max.level*) and the number of the DSS entry (*max.entry*) is kept updated. The role of this register is to help at detecting when sets not currently considered in the DSS should be tracked by it, which happens when their saturation level is below *max.level*. It also indicates where within the DSS a new entry should be inserted.

When the saturation counter of a free set (one that is not associated to another set) is updated, the DSS is checked in case it needs to be updated. The index of this

set is compared in parallel with the indices in the valid entries of the DSS. Under a hit, the corresponding entry is updated with the new saturation level. If this value becomes equal to the displacement limit, the entry is invalidated, since sets with a saturation level larger or equal to this limit are not considered for association. If the set index does not match any entry in the DSS and its saturation level is smaller than *max.level*, this set index and its saturation value are stored in the DSS entry pointed by *max.entry*; otherwise they are dismissed.

Any change or invalidation in the entries of the table of the DSS lead to the update of the *min* and *max* registers. Invalidations take place when the saturation value reaches the displacement limit or when the entry pointed by *min* is used for an association. In this latter case the saturation value of the entry is also set to the displacement limit. This ensures that all the invalid entries have the largest saturation values in the DSS. Thus whenever there is at least an invalid entry, *max* points to it and *max.entry* equals the displacement limit, which is the limit to consider a set for association with a highly saturated set.

The operation of the DSS allows to provide the best candidate for association to a highly saturated set most of the times. The main reason why it may fail to do this is because all its entries may be invalidated in the moment the association is requested. When this happens no association takes place. Obviously, the larger the number of entries in the DSS, the smaller the probability this situation arises. The efficiency of the DSS as a function of its number of entries will be analyzed in Section 2.8.4.

The DSBC has a table with one entry per set called *Association Table* (AT) that stores in the i-th entry $AT(i).index$, the index of the set associated with set i, and a source/destination bit $AT(i).s/\overline{d}$ that indicates in case of being associated, whether the set triggered the association because it became saturated ($s/\overline{d} = 1$) or it was chosen by the Destination Set Selector to be associated because of its low saturation ($s/\overline{d} = 0$). When a set is not associated, its entry stores its own index and $s/\overline{d} = 0$.

2.4.2. Displacement algorithm

Just as in the SSBC, displacements take place when lines are evicted from sets whose saturation counter has its maximum value. In the DSBC, sets are not associated by default to any other specific set, thus another condition for the displacements to take place is that the saturated set is associated to another set. Another important difference with respect to the SSBC is that displacements are unidirectional, that is, lines can only be displaced from the set that requested the association (the one whose counter reached its maximum value), called source set, to the one that was chosen by the Destination Set Selector to be associated to it, which we call destination set. The rationale is that the destination set was chosen among all the ones in the cache to receive lines from the source one because of its low level of saturation. For the same reason, displacements do not depend on the level of saturation of the destination set: once it is designated as destination set, it continues to receive lines displaced from the source until the association is broken. If the same policy as in the SSBC were applied, that is, if displacements only took place when the destination set saturation counter were smaller than K, the average miss rate in our experiments would have been on average 0.6% larger, and the resulting IPC would have been 0.38% worse.

2.4.3. Search algorithm

Just as in the SSBC, there is a displaced bit d per line that indicates whether is has been displaced from another set. The cache always begins a search looking for a line with the desired tag and $d = 0$ in the set with the index i specified by the memory address sought. Simultaneously the corresponding i-th entry in the Association Table, $AT(i)$ is read. Upon a hit, the LRU of the set (and the dirty bit if needed) is modified. Otherwise, the access is known to have resulted in a miss if $AT(i).s/\overline{d} = 0$, as this means that either the set is not associated or this set is the destination set of an association, which cannot displace lines to its associated set. In any case the saturation counter is updated and if it has reached its maximum and the set is not yet associated, a destination set can be requested from the Destination Set Selector while the miss is resolved.

If $AT(i).s/\overline{d} = 1$ the destination set indicated by $AT(i).index$ is searched for an entry with the tag requested and $d = 1$. Here we can get a secondary hit or a definitive miss. In both cases the set saturation counter will be updated, although this will not influence the association. If there is a miss, the LRU line of the destination set will be evicted, and the LRU line from the source set will be moved to the destination set to replace it. This happens in parallel with the resolution of the miss, whose line will be inserted in the source set.

Finally, as it was in the SSBC, DSBC does not swap lines to return them to their original set when they are found displaced in another set.

2.4.4. Disassociation algorithm

The approach followed to break associations is very similar to the one used to avoid unnecessary second searches in the SSBC. A disassociation can take place upon a first search miss (i.e., a native miss) in a destination set i. If the OR of the d bits of this set changes from 1 to 0 as result of the eviction triggered by the miss, the association is broken. This can be calculated once the line to be evicted is decided, as this condition is equivalent to requiring that the OR of the d bits of all the lines but the one to evict is 0. This way, the detection of the disassociation and the changes it involves take place in parallel with the eviction itself and the resolution of the miss. The disassociation requires accessing the AT of the source set of the association, as provided by $AT(i).index$, and clearing the association there. The entry for the destination set is then also modified setting $AT(i).index = i$.

2.4.5. Discussion

Figure 2.3 shows an example of the DSBC operation with the same references and a cache with the same parameters as in Figure 2.2. The first reference is mapped to set 0, where a miss occurs. Since this set is not associated ($AT(0).index = 0$) but its saturation counter has its maximum value, a destination set for an association is requested. The figure assumes the Destination Set Selector provides set 3 as candidate, proceeding then to evict the LRU line in set 3 to replace it with the LRU line in set 0. When the missed line arrives from memory it is stored in the block

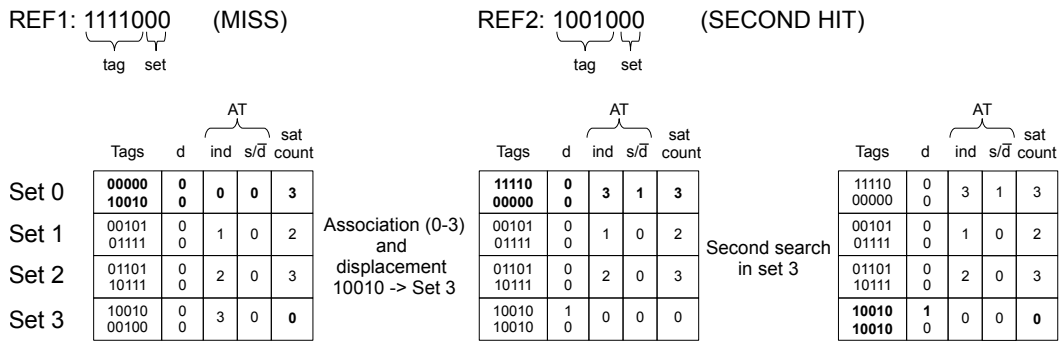


Figure 2.3: Dynamic SBC operation in a 2-way cache with 4 sets. The upper tag in each set is the most recently used and the saturation counters operate in the range 0 to 3.

that has been made available in set 0. The second reference is mapped again to the set 0 resulting in a miss. A second search is initiated in set $AT(0).index = 3$, where it is found.

The greater flexibility of the DSBC allows it to apply a more aggressive displacement policy, as Section 2.4.2 explains. Section 2.6 will show it also achieves better results. Beyond performance measurements, graphical representations also help explain the net effect of SBC on a cache. Figure 2.4 illustrates it showing the distribution of the saturation level across the sets of the L2 cache of the two-level cache configuration of Table 2.1 during part of the execution of the *omnetpp* benchmark of the SPEC CPU 2006 suite. The level is measured with a saturation counter in the range 0 to 15. The baseline in Figure 2.4 (a) has a high ratio of highly (level 11 to 15) and lowly (level 0 to 5) saturated sets. The SSBC in Figure 2.4 (b) basically turns highly-saturated sets into medium-saturated sets. The DSBC in Figure 2.4 (c) alleviates more highly-saturated sets without generating medium-saturated sets.

2.5. Simulation environment

To evaluate our approach we have used the SESC simulator [58] with two baseline configurations, one with two on-chip cache levels and another one with three. Both configurations, detailed in Table 2.1, are based on a four-issue CPU clocked at 4GHz

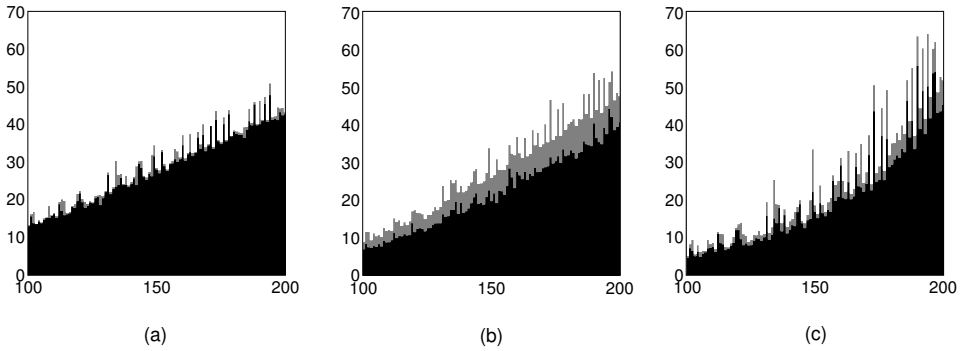


Figure 2.4: Distribution of the sets with a high saturation level(black), medium saturation level(gray) and low saturation level(white) during a portion of the execution of omnetpp in the L2 cache of the two-level configuration. (a) Baseline (b) Static SBC (c) Dynamic SBC. Samples each $5 * 10^5$K accesses.

with an hybrid branch predictor [49]. The tag check delay and the total round trip access are provided for the L2 and L3 to help evaluate the cost of second searches when the SBC is applied. Our three-level hierarchy is somewhat inspired in the Core i7 [24], the L3 being proportionally smaller to account for the fact that only one core is used in our experiments. Both configurations allow an aggressive parallelization of misses, providing between 16 and 32 Miss Status Holding Registers per cache. As in several existing processors [11][79], and works in the bibliography [10][18][55], the accesses to non-first level caches access sequentially the tag and the data arrays. This reduces the power dissipation of large cache arrays and limits the additional delay of second searches to the tag check delay. We have used CACTI [21] to derive the latency related to each component of the memory hierarchy.

We use 10 representative benchmarks of the SPEC CPU 2006 suite, both from the INT and FP sets. They have been executed using the reference input set (*ref*), during 10 billion instructions after the initialization. Table 2.2 characterizes them providing the number of accesses to the L2 during the 10^{10} instructions simulated, the miss rate in the L2 cache both in the two-level (2MB L2) and the three-level (256kB) configurations, and whether they belong to the INT or FP set of the suite.

Table 2.1: Architecture. In the table RT, TC and MSHR stand for round trip, tag directory check and miss status holding registers, respectively.

Processor	
Frequency	4GHz
Fetch/Issue	6/4
Inst. window size	80 int+mem, 40 FP
ROB entries	152
Integer/FP registers	104/80
Integer FU	3 ALU,Mult. and Div.
FP FU	2 ALU, Mult. and Div.
Common memory subsystem	
L1 i-cache & d-cache	32kB/8-way/64B/LRU
L1 Cache ports	2 i/ 2 d
L1 Cache latency (cycles)	4 RT
L1 MSHRs	4 i / 32 d
System bus bandwidth	10GB/s
Memory latency	125ns
Two levels specific memory subsystem	
L2(unified) cache	2MB/8-way/64B/LRU
L2 Cache ports	1
L2 Cache latency (cycles)	14 RT, 6 TC
L2 MSHR	32
Three levels specific memory subsystem	
L2(unified) cache	256kB/8-way/64B/LRU
L3(unified) cache	2MB/16-way/64B/LRU
Cache ports	1 L2, 1 L3
L2 Cache latency (cycles)	11 RT, 4 TC
L3 Cache latency (cycles)	39 RT, 11 TC
MSHR	32 L2, 32 L3

Table 2.2: Benchmarks characterization. MR stands for miss rate.

Bench	**L2 Accesses**	**2MB L2 MR**	**256kB L2 MR**	**Comp.**
bzip2	125M	9%	41%	INT
milc	255M	71%	75%	FP
namd	63M	2%	5%	FP
gobmk	77M	5%	10%	INT
soplex	105M	8%	15%	FP
hmmer	55M	10%	41%	INT
sjeng	32M	26%	27%	INT
libquantum	156M	74%	74%	INT
omnetpp	100M	28%	91%	INT
astar	192M	23%	48%	INT

It is a mix of benchmarks that vary largely both in number of accesses that reach the caches under the first level and in miss ratios in the L2 cache.

2.6. Experimental evaluation

The SBC has been applied, for both the static and the dynamic version, in the second level for the two-level configuration and in the two lower levels for the three-level configuration. The dynamic SBC uses a Destination Set Selector (described in Section 2.4.1) with four entries based on our experiments (in Section 2.8.4).

Figure 2.5 shows the ratio of accesses that result in a miss, a hit, and a secondary hit in the L2 and L3 caches in the two memory hierarchies tested, using standard caches, SSBC, and DSBC for each one of the benchmarks analyzed. The last group of columns (mean), represents the arithmetic mean of the rates observed in each cache. We can see that the SBCs basically keep the same ratio of first access hits as a standard cache, and they turn a varying ratio of the misses into secondary hits. When the baseline miss rate is small or there are few accesses, the SBCs seldom perform displacements of lines and second searches happen also infrequently. Also, the DSBC achieves better results than the SSBC, as expected.

Figures 2.6 (a) and 2.6 (b) show the performance improvement in terms of instructions per cycle (IPC) for each benchmark in the two-level and the three-level configurations tested, respectively. The figures compare the baseline not only with the SSBC and the DSBC, but also with the baseline system where the L2 and the L3 have duplicated their associativity. This latter configuration is tested to show the difference between associating two sets of K lines following the SBC strategy and using sets of $2K$ lines. The bar labeled *geomean* is the geometric mean of the individual IPC improvements seen by each benchmark.

In the two-level configuration the SBC always has a positive or, at worst, negligible effect on performance. Two kinds of benchmarks get no benefit from the SBC: those with a small miss rate, like 444.namd or 445.gobmk, in which our proposal can do little to improve an already good cache behavior; and 458.sjeng, which has

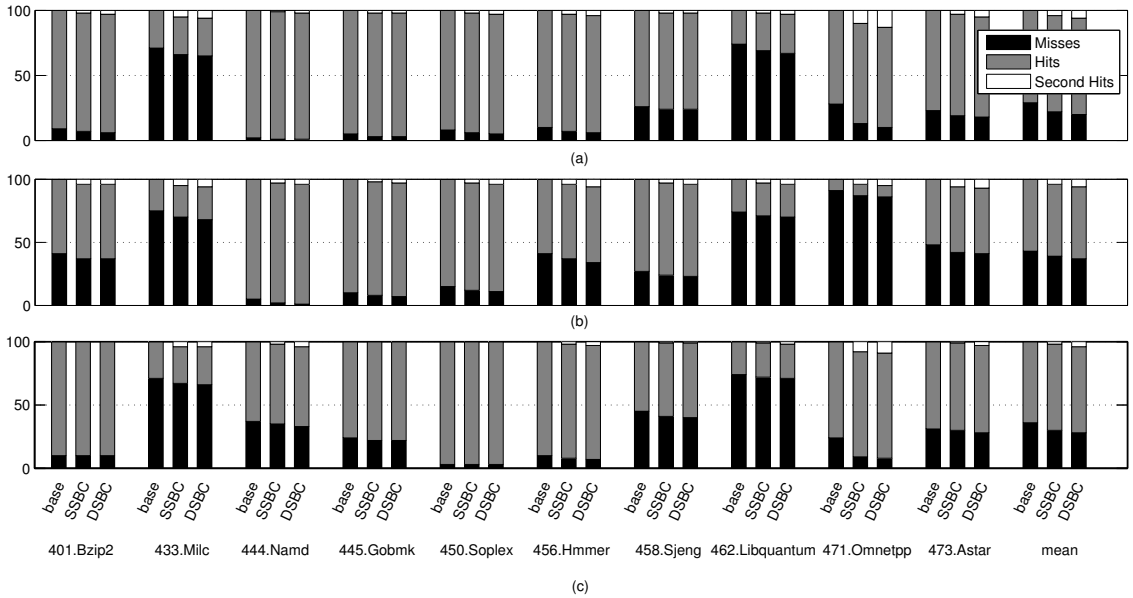


Figure 2.5: Miss, hit and secondary hit rates for the (a) L2 cache in the two-level configuration, (b) L2 cache in the three-level configuration, and (c) L3 cache in the three-level configuration.

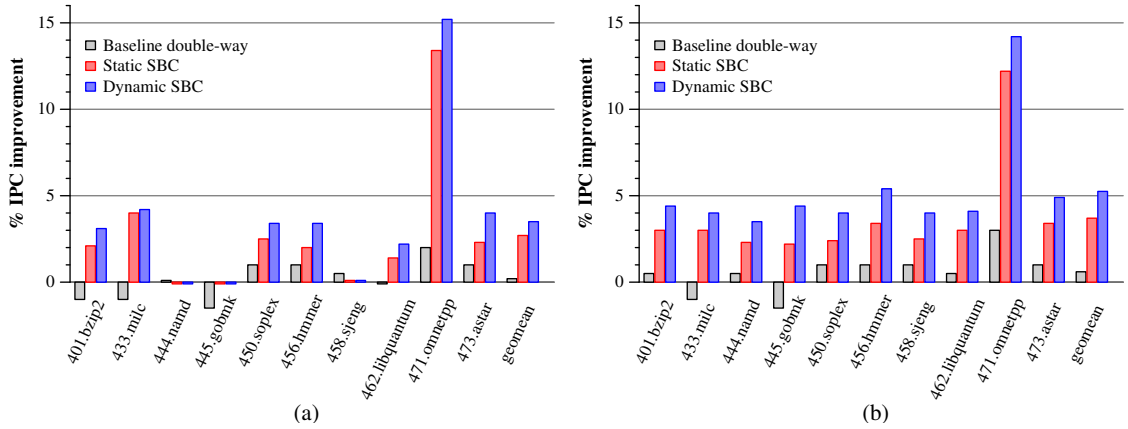


Figure 2.6: Percentage IPC improvement over the baseline in the two-level (and) and the three-level configuration (b) duplicating the L2 and L3 associativity or using SBC in both levels.

very few accesses to the L2, just 3.2 each 1000 instructions, as Table 2.2 shows. The small number of accesses reduces the influence of the L2 behavior in the IPC, and more importantly it reduces the frequency of triggering of the SBC mechanisms.

In the three-level configuration the improvement is larger and applies to all the benchmarks. The benchmarks that did not benefit from the SBC in the two-level configuration benefit now for two reasons. One is the larger local miss ratios either in the L2 or in the L3. The other is that in this 256 kB L2 cache (modeled after the one in the Core i7) the accesses are spread on 8 times less sets than in the 2MB cache of the two-level configuration. This increases the working set of each set, generating more SBC-specific activity. The DSBC systematically outperforms the SSBC, which in its turn achieves much better results than duplicating the associativity of the caches. Since the SSBC associates exactly the two same sets that a duplication of the associativity merges, these results outline the benefit of sharing resources among sets under the control of a policy that triggers this sharing only when it is likely it is going to be beneficial and disables it when the feedback is not good.

We compare next the performance of the SBC in terms of miss rate reduction with the V-Way cache described in Section 2.2, which requires an additional 11% storage and area overhead on the L2 cache of our two-level baseline configuration, and the Dynamic Insertion Policy (DIP) [53], because its cost also scales well with the cache size. DIP is a proposal to adapt dynamically the policy of insertion of new lines in sets, alternating between marking the most recently inserted lines in a set as most recently used lines (the traditional policy) or the least recently used ones, the replacement policy being LRU. Notice that if the latter case, only if the block is accessed again in the cache will it become the MRU in its set. Otherwise the next miss will trigger its eviction. This system helps keep the most important part of the working set in the cache when the size of this set is much larger than the cache. We have not made performance comparisons with Scavenger because its large hardware requirements would make the comparison very unfair. Scavenger requires more than 12% additional storage, in comparison with the 0.28% and 0.55% that SSBC and DSBC have, as we will see in Section 2.7. Something similar happens with the area required, estimated at more than 12% for Scavenger, while it is below 1% for the SBC (Table 2.4).

Figure 2.7 compares the miss rates among SSBC, DSBC, V-Way cache and DIP

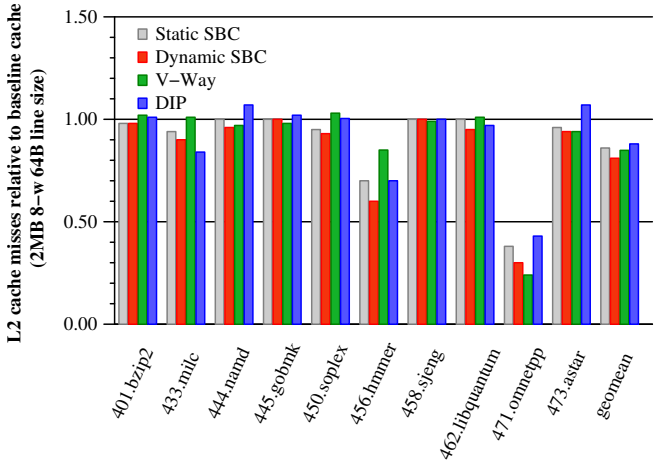


Figure 2.7: Comparison with recent proposals in terms of number of cache misses relative to the L2 cache of our two-level configuration.

in the L2 cache for the two-level configuration used previously. Data shown are relative to miss rate of the baseline configuration. DIP has been simulated with 32 dedicated sets and $\epsilon = 1/32$ (see [53]). The last group of bars correspond to the geometric mean of the ratios of reduction of the miss rate for the four policies. The results vary between the 19% reduction for the dynamic SBC and the 12% reduction for DIP, which is the simplest and cheapest alternative. The V-way cache achieves a 15% reduction, slightly better than the 14% one of the static SBC. Benchmark by benchmark, the V-way cache is the best one in three of them, DIP in one, and the dynamic SBC in the other six ones. We must take into account that DIP and the V-Way cache turn misses into hits, while the SBC turns them into secondary hits, which suffer the delay of a second access to the tag array. On the other hand, the duplication of tag-store entries, the addition of one pointer to each entry and a MUX to choose the correct pointer increases the V-Way tag access time around 39%, while the SBC has very light structures (up to 17 bits per set plus one bit per tag-store entry), thus having a negligible impact on access time.

2.6.1. Average memory latency and power consumption

As we just said in the paragraph above, hit and miss rates are not the best characterization for SBCs because they involve second searches that make secondary

hits more expensive than first hits, and which delay the resolution of misses that need the second search to be confirmed. The advantages of SBC are better measured in Figure 2.8, which shows the average data access time improvement of the static and dynamic SBC with respect to the baseline caches for each benchmark.

Despite the overhead of the second searches, the SBC almost never increases the average access time of any benchmark. There is only a small 1% slowdown in the L2 cache in the two-level configuration for 444.namd and 445.gobmk, because their second searches contribute very little to reduce its already minimal miss rate. Not surprisingly the greater flexibility of the DSBC allows it to choose better suited cache sets for the displacements than the SSBC, leading to better average access times. The average improvement (geometric mean) of the access time in the L2 of our two-level configuration is 4% and 8% for the SSBC and the DSBC, respectively. In the three-level configuration the average reduction is 3% and 6% for the L2, and 10% and 12% for the L3, for the SSBC and the DSBC, respectively. These results translated into power consumption reductions of 9% and 11% for SSBC and DSBC in the two-level baseline configuration, shown in Figure 2.9. Finally, Figure 2.10 shows how SSBC and DSBC reduce the power consumption of the memory hierarchy in the three-level configuration by 10% and 12% with respect to the baseline, respectively. Note that these two last figures also show the impact of the accesses to each level of the memory hierarchy in the total power consumption.

2.7. Cost

In this section we evaluate the cost of the SBC in terms of storage requirements, area and energy, which has been estimated using CACTI [21].

The SBC requires additional hardware because of the need of a saturation counter per set to monitor its behavior and additional bits in the directory to identify displaced lines (d bit). The SSBC has an additional bit per set to know whether second searches are required. The DSBC instead requires an Association Table with one entry per set that stores a $s/\overline{d}$ bit to specify whether the set is the source or the destination of the association, and the index of the set it is associated to. It also requires a Destination Set Selector (DSS) to choose the best set for an association,

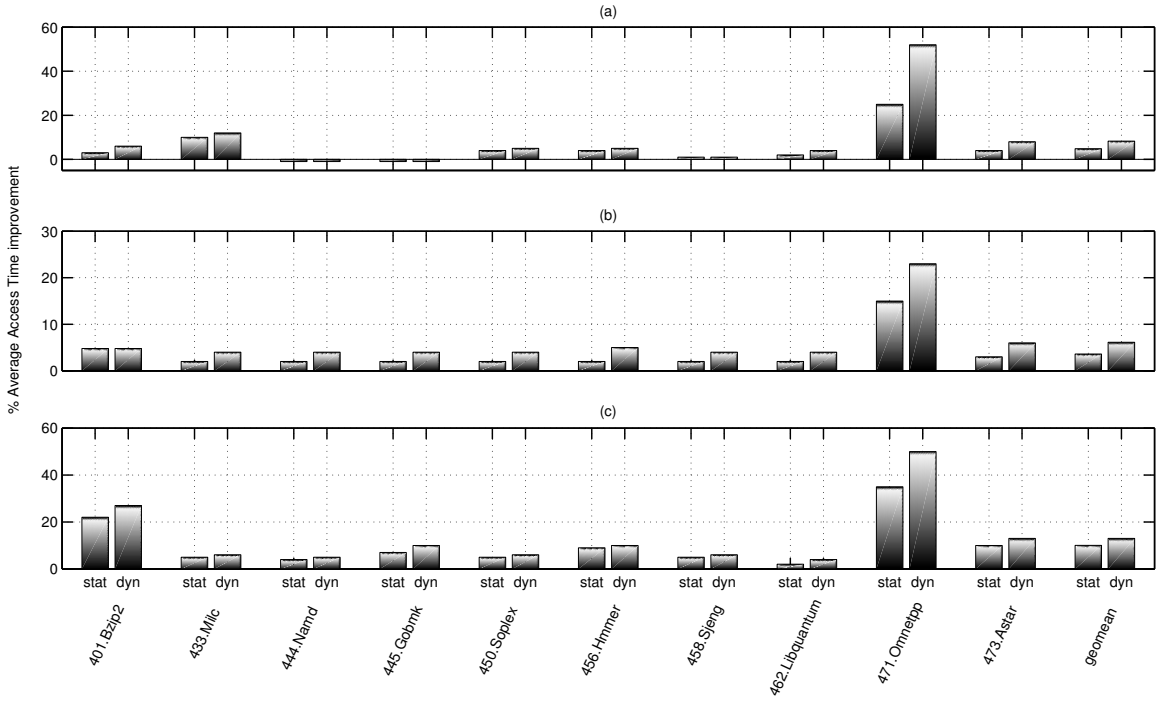


Figure 2.8: Average access time reduction achieved by the static and the dynamic SBC in the (a) L2 cache in the two-level configuration, (b) L2 cache in the three-level configuration, and (c) L3 cache in the three-level configuration.

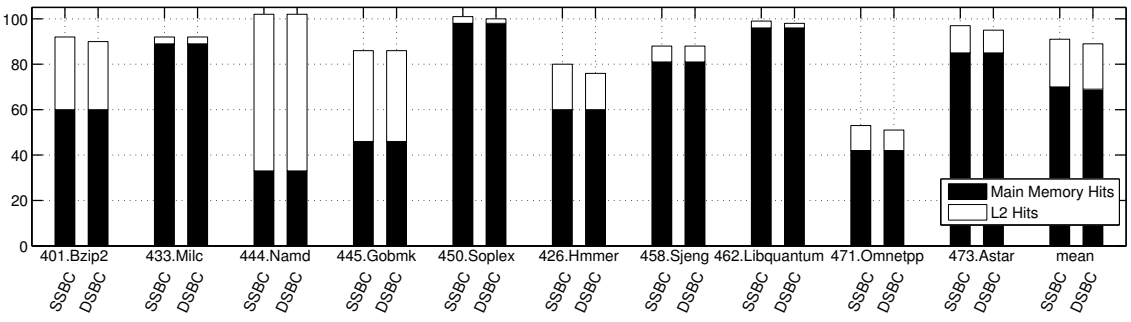


Figure 2.9: Average power consumption achieved by the SBC related to the baseline and breakdown of the accesses in the **two-level** configuration.

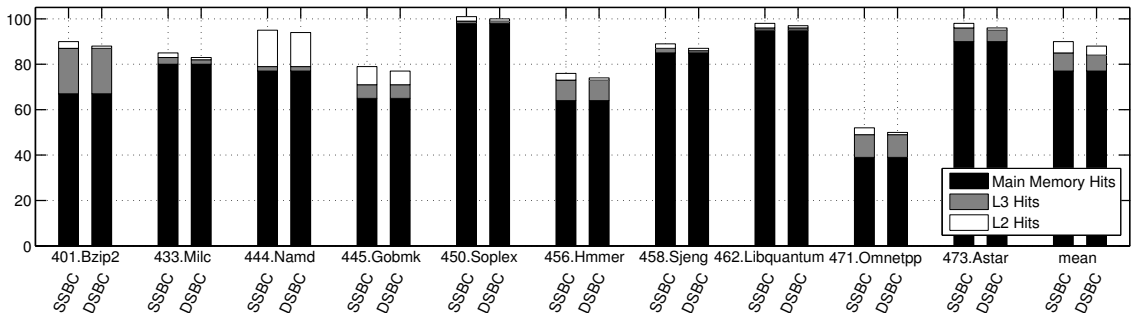


Figure 2.10: Average power consumption achieved by the SBC related to the baseline and breakdown of the accesses in the **three-level** configuration.

a 4-entry DSS being used in our evaluation. Based on this, Table 2.3 calculates the storage required for a baseline 8-way 2 MB cache with lines of 64B assuming addresses of 42 bits. As we can see, the SSBC and the DSBC only have an overhead of 0.31% and 0.58% respectively, compared to the baseline configuration. The energy consumption overhead on average per access calculated by CACTI is less than 1% for SBC and 79% for the baseline with double associativity, and the corresponding area overhead is shown in Table 2.4. We see that the SBC not only offers more performance, but also requires less energy and area than duplicating the associativity.

Table 2.3: Baseline and SBC storage cost in a 2MB/8-way/64B/LRU cache. B stands for bytes.

	Base	Static SBC	Dynamic SBC
Tag-store entry:			
State(v+dirty+LRU+[d])	5 bits	6 bits	6 bits
Tag ($42 - \log_2 sets - \log_2 ls$)	24 bits	24 bits	24 bits
Size of tag-store entry	29 bits	30 bits	30 bits
Data-store entry:			
Set size	512B	512B	512B
Additional structs per set:			
Saturation Counters	-	4 bits	4 bits
Second search bits	-	1 bit	-
Association Table	-	-	12+1 bits
Total of structs per set	-	5 bit	17 bits
DSS (entries+registers)	-	-	10B
Tag-store entries	32768	32768	32768
Data-store entries	32768	32768	32768
Number of Sets	4096	4096	4096
Size of the tag-store	116kB	120kB	120kB
Size of the data-store	2MB	2MB	2MB
Size of additional structs	-	2560B	8714B
Total	**2164kB**	**2170kB** (0.28%)	**2176kB** (0.55%)

Table 2.4: Baseline and SBC area. Percentages in the Total column are related to the Baseline configuration.

Configuration	Components	Details	Subtotal	Total
Baseline	Data + Tag	2MB 8-way 64B line size + tag-store	12,57 mm^2	**12,57** mm^2
Baseline with double associativity	Data + Tag	2MB 16-way 64B line size + tag-store	14,52 mm^2	**14,52** mm^2 (> 3%)
Static SBC	Data + Tag	2MB 8-way 64B line size + tag-store (with additional d bit)	12,58 mm^2	**12,60** mm^2 (< 1%)
	Counters	4096*4 bits	0,01 mm^2	
	Second search bits	4096 bits	$< 0,01$ mm^2	
Dynamic SBC	Data + Tag	2MB 8-way 64B line size + tag-store (with additional d bit)	12,58 mm^2	**12,64** mm^2 (< 1%)
	Counters	4096*4 bits	0,01 mm^2	
	Association Table	4096*12 bits	0,04 mm^2	
	DSS (entries + regs)	4*(1+12+4)+2*(2+4) bits	$< 0,01$ mm^2	

2.8. Analysis

In this section we evaluate how the performance and cost of the SBC vary with respect to the parameters of the cache. We also analyze how it compares to the usage of a victim cache whose cost is comparable to the overhead of the SBC and we study its internal behavior. Finally, we profile the DSS efficiency. All along this section we will always use as baseline the 2MB/8-way/64B/LRU L2 cache of our two-level configuration.

2.8.1. Impact of varying cache parameters

Table 2.5 shows the miss rate reduction achieved by the static and the dynamic SBC as well as the storage overhead they involve as the cache size varies between 256kB and 4MB. Both kinds of SBC always reduce the average miss rate obtained, but as the cache size increases the working set of some benchmarks fits better, reducing the opportunities of improving it.

Table 2.6 studies the cost-benefit of both SBC proposals comparing the miss rate reduction achieved by them versus the additional storage cost they incur as a function of the line size in the baseline cache. The increase in the line size reduces proportionally both the number of lines and sets, being the SBC cost mostly proportional to the latter as we can see. The reduction of the number of sets and the fact their lines keep more data also makes more probable the SSBC finds the static pairs of sets it is able to associate are too saturated to trigger displacements. The greater flexibility of the DSBC allows to overcome better this problem. This is why the DSBC behaves better than the SSBC as the line size increases.

Table 2.7 makes the same study from the point of view of the associativity considering values of 8, 16 and 32. The increase of associativity reduces the number of sets, and thus the relative cost of the SBC, but increases the tag size. Miss rates and their reduction stay very flat. Also, just as the experiments in Section 2.6 considering caches that duplicated the associativity, this table shows that making shared usage of the lines of two sets under heuristics like the ones proposed by the

Table 2.5: Cost-benefit analysis of the static and the dynamic SBC as a function of the cache size.

Cache size	Baseline miss rate	SSBC miss rate	DSBC miss rate	SSBC miss rate reduction	DSBC miss rate reduction	SSBC storage overhead	DSBC storage overhead
256KB	45.13%	40.81%	39.24%	9.6%	13.1%	0.26%	0.48%
512KB	39.07%	35.54%	34.47%	9.2%	11.8%	0.27%	0.50%
1MB	33%	30.84%	29.14%	6.55%	9.3%	0.28%	0.52%
2MB	25.6%	23.25%	22.3%	9.2%	12.8%	0.28%	0.55%
4MB	20.7%	19.6%	19.3%	5.4%	6.8%	0.29%	0.57%

Table 2.6: Cost-benefit analysis of the static and the dynamic SBC as a function of the line size.

Line size	Baseline miss rate	SSBC miss rate	DSBC miss rate	SSBC miss rate reduction	DSBC miss rate reduction	SSBC storage overhead	DSBC storage overhead
64B	25.6%	23.25%	22.31%	9.2%	12.8%	0.28%	0.55%
128B	27.46%	25.6%	25%	6.5%	9%	0.15%	0.27%
256B	24.6%	23.2%	22.3%	5.7%	9.3%	0.07%	0.13%

Table 2.7: Cost-benefit analysis of the static and the dynamic SBC as a function of the associativity.

Associativity	Baseline miss rate	SSBC miss rate	DSBC miss rate	SSBC miss rate reduction	DSBC miss rate reduction	SSBC storage overhead	DSBC storage overhead
8-ways	25.6%	23.25%	22.31%	9.2%	12.8%	0.28%	0.55%
16-ways	25.1%	22.8%	21.88%	9.2%	12.83%	0.24%	0.36%
32-ways	24.6%	22.28%	21.43%	9.4%	12.88%	0.21%	0.27%

SBC is much more effective than organizing the cache lines in sets with twice or even four times larger. This way, even the 8-way static and dynamic SBC have 5.5% and 9.31% less misses than the 32-way baseline, respectively.

2.8.2. Victim cache comparison

We compare here the SBC performance with that one of two victim caches in the 2MB/8-way/64B/LRU L2 cache of our two-level configuration baseline. Figure 2.11 (a) shows a comparison of the L2 cache miss rates among a static SBC, a dynamic SBC, and the cache extended with a fully-associative victim cache [34] of either 8kB or 16kB of data store, relative to the L2 two-level baseline configuration. We have chosen these sizes because as Table 2.3 shows, the storage overhead for the L2

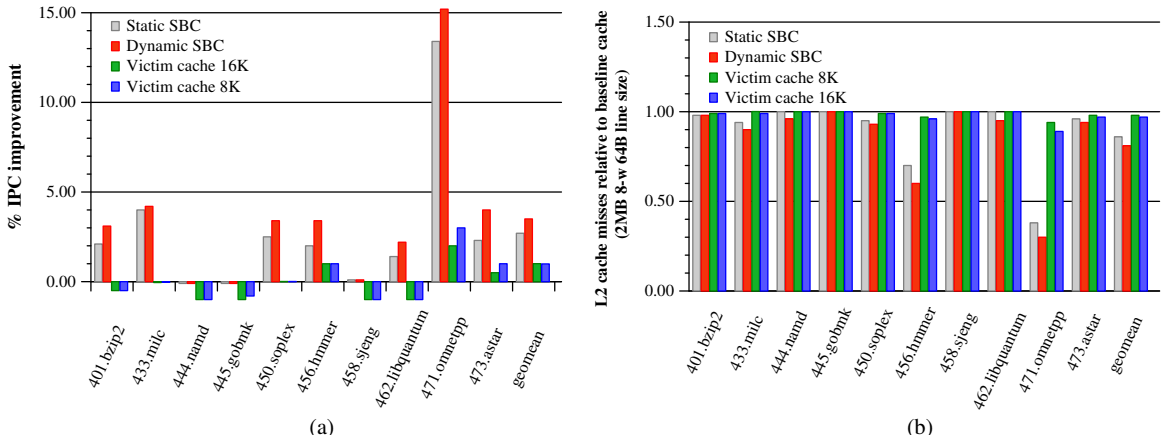


Figure 2.11: Comparison of the static SBC, the dynamic SBC, a victim cache of 8kB and a victim cache of 16kB in terms of IPC (a) and miss rate (b) relative to the one in the L2 in the two-level baseline configuration.

cache configuration considered is about 7kB for the static SBC and about 13 kB for the dynamic SBC. Thus, the 8kB and the 16kB victim caches are larger than the static and the dynamic SBC respectively. If their tag-store were considered too, they would be even more expensive in comparison. We see how with less resources, any SBC performs better than the largest victim cache. Figure 2.11 (b) makes the same comparison based on the IPC.

2.8.3. SBC behavior

While comparisons in miss rate, average access time or IPC allow to assess the effectiveness of the SBC with respect to other designs, measurements on its internal behavior allow to understand better how it achieves these results. Thus, we analyze here this behavior based on measurements in the L2 cache of our two-level configuration. This way, the hit rate observed in the second searches, that is, the ratio of second searches that result in a secondary hit, is on average 36.3% and 47.7% for the SSBC and DSBC, respectively. The SSBC is more conservative in displacing lines than the DSBC because of its restriction on the associated set. As a result, less lines are displaced, leading to a smaller second access hit ratio. In fact the SSBC displaces an average of 1.7 lines per association (i.e. since the *sc* bit in

an association is activated until it is reset), while the DSBC displaces an average of 2.15 lines before the association is broken. On the other hand, the conservative policy of the SSBC leads it to make safer decisions than the DSBC on which lines it is interesting to displace to the associated sets, that is, the lines it displaces are more likely to be referenced again. The result of this is that the average number of secondary hits per line displaced is 3.64 in the SSBC, while it decreases to 3.29 in the DSBC.

It is also interesting to examine the frequency of second searches, as they may generate contention in the tag-array. On average only 10.3% and 10.2% of the accesses to the cache require second searches in the SSBC and the DSBC, respectively.

2.8.4. Destination Set Selector efficiency

A request for a destination set made to the Destination Set Selector (DSS) may result in four outcomes. If the DSS provides a candidate, this cache set can (A) actually have the smallest level of saturation among the available sets in the cache or (B) not. The DSS will not provide a candidate if all its entries are invalid. This may happen either because (C) there are actually no candidates in the cache (all the sets are either associated or too saturated), or (D) there are candidates in the cache, but not in the DSS. Figure 2.12 shows the evolution of the average percentage of times each one of these four situations happens during the execution of our benchmarks in the L2 cache of the two-level configuration as the number of entries in the DSS varies from 2 to 128. The outcomes are labeled A, B, C and D, following our explanation. We see that even with just two entries the DSS has a quite good behavior, since outcomes A and C, in which the DSS works as well as if it were tracking the behavior of all the sets, add up to 80%. With 4 entries A+C behavior improves to 90%, and after that there is a slow slope until almost 100% of the outcomes are either A or C with 128. Based on this, a 4-entry DSS has been chosen for all the evaluations of the SBC shown in the other sections and chapters of this dissertation, as this number optimizes the balance between hardware and power required and benefit achieved. In this graph we can also see that under the conditions requested in the DSBC, around 35% of the association requests are satisfied.

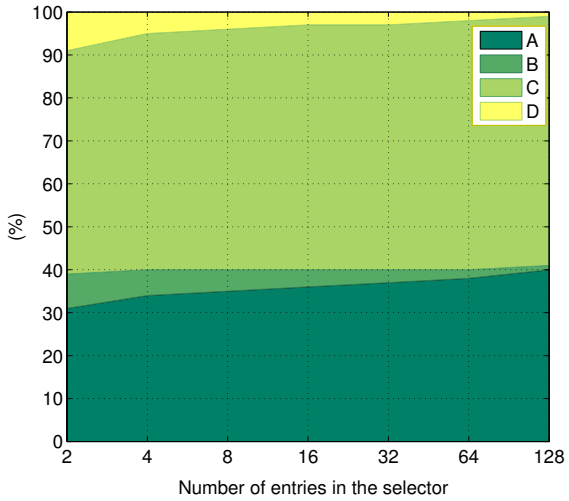


Figure 2.12: Percentage of association requests made to the Destination Set Selector (DSS) in the L2 cache of the two-level configuration that (A) are satisfied with a set with the minimum level of saturation, (B) are satisfied with a set whose level of saturation is not the minimum available, (C) are not satisfied because there are no candidate sets in the cache, and (D) are not satisfied because none of the existing candidate sets is in the DSS, depending on the number of entries in the DSS.

2.9. Summary

This chapter proposes the Set Balancing Cache (SBC), a new design aimed at non-first level caches with a good cost-benefit relation. This cache associates sets with a high demand with sets that have underutilized lines in order to balance the load among both kinds of sets and, thus, to reduce the miss rate. The *Set Saturation Level*, which measures the degree to which a set is able to hold its particular working set, is introduced. This value, which indicates the degree of pressure on a set, is provided by a counter per set called saturation counter which is increased under a miss and decreased under a hit. The balance is materialized in the displacement of lines from cache sets with a high level of saturation to sets that seem to be underutilized, the displaced lines being found in the cache in subsequent searches. Two designs have been presented: a static one, which only allows displacements between preestablished pairs of sets, and a dynamic one that tries to associate each highly saturated set with the least saturated cache set available. The selection of this least saturated set is made by a very cheap hardware structure called Destination

Set Selector (DSS), which yields near-optimal selections.

Experiments using representative benchmarks achieved an average reduction of 9.2% and 12.8% of the miss rate for the static and the dynamic SBC, respectively, or 14% and 19% computed as the geometric mean.

This led to average IPC improvements between 2.7% and 5.25% depending on the type of SBC and the memory hierarchy tested. Furthermore, the SBC designs proved consistently to be better than increasing the associativity, both in terms of area and performance.

Chapter 3

Bimodal Set Balancing Cache

3.1. Introduction

We have studied in the previous chapter the problem of the lack of uniformity in the distribution of memory references among the sets of set-associative caches, which is a fundamental source of conflict misses. The load balancing between sets that SBC provides can reduce not only conflict misses, but also capacity misses. The reason is that SBC tries to displace lines in the working set (i.e. live lines) to take the place in the cache of lines that are not in use (dead lines), even when those dead lines could have been accessed more recently. This avoids misses that would even happen with full associativity. Still, SBC cannot help for example in situations of thrashing, so it does not suffice to deal with all the capacity misses. Some recent proposals seem more adequate to reduce this kind of misses. A very good example is [53], which targets memory-intensive workloads with working sets that do not fit in the cache for which the traditional LRU replacement policy is counterproductive.

In this chapter we introduce the Bimodal Set Balancing Cache or BSBC, a technique which extends the Set Balancing Cache (SBC), which can do little to improve the performance when the working set of a workload is larger than the cache size, by complementing with a policy particularly suitable to reduce capacity misses. If the lack of lines to hold the working sets persists after the displacement of lines from oversubscribed sets to underutilized ones performed by the SBC, the BSBC

applies a policy to address problems of capacity. Namely, the insertion policy of highly saturated sets is changed to the Bimodal Insertion Policy (BIP) [53], which often inserts lines in the LRU position instead of the MRU one. This avoids that lines that are dead on arrival expel other lines from the cache as they descend in the LRU stack. Furthermore, [53] also introduces the Dynamic Insertion Policy (DIP) policy, which chooses dynamically between the traditional insertion policy and BIP based on a set dueling mechanism that tries to choose the one that incurs fewer misses. Our experiments show that our coordinated approach to reduce conflict and capacity misses works substantially better than simply applying simultaneously DSBC and DIP in a cache.

3.2. Background and Motivation

The related proposals discussed in Section 2.2 emphasize the flexibility of placement of lines in the cache to improve miss rates or access time. Other researchers have focused on keeping in the cache the most useful lines. A first example of this kind of works are the adaptive insertion policies in [53], which will be thoroughly revised further in this section due to its significance in the cache design presented in this chapter. Another proposal in this direction are the pseudo-LIFO replacement policies [9]. The probabilistic escape LIFO, which dynamically learns the best eviction positions within the fill stack and prioritizes the ones close to the top of the stack, belongs to this latter family. This family of policies relies on fill stack, where lines once are installed in the cache can only go downwards, instead of the traditional recency one. Then, the degree of reuse of lines beyond each stack position is estimated by means of a dynamic mechanism and evictions are performed from several escape points within the stack. Lines with high eviction priorities are those residing close to the top of the stack while those lines which are likely to be reused reside at the bottom.

More recently, DRRIP [28] chooses the most appropriate Re-Reference Interval Prediction (RRIP) insertion policy for each application based on set dueling. The competing insertion policies are the Static RRIP, which inserts new lines with a long re-reference interval prediction, and the Bimodal RRIP, which inserts most of the new lines with a distant value; such value making the new line become an eligible

victim in the next replacement operation. When a hit occurs in a certain line its re-reference interval prediction is reduced, while every miss increases the re-reference interval prediction of each line in the set until a victim is eventually found.

All techniques presented so far mainly tackle only one kind of problem. We have combined two techniques in order to deal with both conflict and capacity misses at the same time. For this purpose, we have extended the Dynamic Set Balancing Cache with the possibility of applying the novel insertion policies proposed in [53]. Both approaches will be discussed in turn, followed by a constructive critic of their limitations and their complementarity.

The basic idea of the Dynamic Set Balancing Cache or DSBC, as we could see in Chapter 2, is to alleviate the problems of oversubscribed cache sets by moving part of the lines originally mapped to them to other sets that have underutilized lines. This requires detecting the degree to which each cache set is able to hold its working set. The DSBC achieves this with the Set Saturation Level (SSL) metric, which is tracked separately for each set by means of a saturation counter. Still SBC can do little for example in situations of thrashing, so it does not suffice to deal with all the capacity misses.

As we previously stated in Section 1.1, most caches nowadays use a LRU replacement policy in which lines are inserted in the MRU position in the recency stack. The lines must then descend this stack position by position until they reach the LRU position, which is the one of the lines evicted under a miss in the set. Although this policy works very well for many workloads, in [53] it was observed that it often leads memory-intensive workloads with working sets larger than the cache size to thrashing. As a result, they proposed an LRU Insertion Policy (LIP) which always inserts lines in the LRU position of the recency stack. If the inserted line is reutilized, it is moved to the MRU position, as in any cache. If the line is not reused before the next miss in the set, the next line inserted replaces it. At this point it is very important to remember that [53], just as [60] and this Thesis, deal with non first level caches. LIP exploits the fact that in these caches many lines are dead on arrival (i.e. they are not reused in the cache before their eviction) because all their potential short term temporal or spatial locality is exploited in upper level caches.

While LIP works well for some workloads, it may tend to retain in the non-LRU positions of the recency stack lines which are actually not useful, that is, that do not belong to the current working set. Thus a Bimodal Insertion Policy (BIP) that tries to adapt the contents of the set to the active working set of the application was also proposed in [53]. BIP achieves this by inserting with a low probability ε the incoming lines in the MRU position of the recency stack, operating like LIP all the other times.

There is not an absolute winner between BIP and the traditional MRU insertion policy, BIP being better suited for some applications and the traditional policy for others. Thus [53] proposed the Dynamic Insertion Policy (DIP), which uses set dueling to track the behavior of both insertion policies in order to apply the best one to the remaining cache sets, called follower sets. Set dueling requires dedicating a fraction of the cache sets to operate always under BIP and another fraction to apply always MRU insertion. A group of 32 sets dedicated to each policy is found to be good in [53]. The misses in one of the groups of sets increase the value of a global saturating counter while the misses in the other group decrease it. The most significant bit of the counter indicates then which is the best policy in each moment. All the follower sets apply the policy indicated by the counter.

The DSBC relies on the unbalance of the working set of different sets to trigger the mechanisms that make it different from a standard cache. As a result it is oriented to reduce conflict misses rather than capacity misses. Another consequence of this approach is that its metrics must be defined at the cache set level in order to find these unbalances, which is indeed the case of the SSL.

The adaptive insertion policies proposed in [53] alleviate the lack of capacity of the cache to hold the data set manipulated by an application. Since this is a global problem for all the cache, these policies are applied to all the sets at once. For the same reason, DIP, the one that can adapt dynamically to the characteristics of the workloads, relies on a global metric gathered on the behavior of all cache sets.

This way, it looks straightforward that DSBC and DIP should be complementary and that implementing them simultaneously in a cache will offer a higher level of protection against misses than using only one of them. This is also very feasible given their reduced hardware overhead. Nevertheless, the simultaneous implementation

of DSBC and DIP, referred as DSBC+DIP henceforth, yields results very similar or even worse than the ones achieved with any of them independently, as we will see in Section 3.5. The main problems happen when DIP chooses BIP for the followers. DSBC displaces the LRU line of the source sets, as it seems the natural option. This means that the line that DIP exposes to be evicted on the next miss in the source set of an association is actually saved by DSBC, which moves it to the destination set. Since these lines are not actually useful, their existence in the destination set gives place to unsuccessful second searches, which delay the resolution of the miss by the time taken to make a new access to the tag array. Even worse, they may expel an actually useful line just inserted in the LRU position of the destination set before it gets a chance to be reused.

When DIP chooses the traditional insertion policy for the followers, a DSCB+DIP cache behaves very much as a DSBC with two penalties. The first one, which is inherent to DIP, is the existence of sets (32 in our implementation, as advised in [53]) that are forced to apply BIP for the sake of the set dueling even when it is not performing well. The other is that when these sets are involved in an association they generate the problems discussed above.

As a result, while these policies seem complementary, they require a coordinated approach to work properly together. Our proposal is presented in the next section.

3.3. Bimodal Set Balancing Cache

A first issue we explore in the attempt to exploit jointly DSBC and the adaptive insertion policies is the possibility of using the same metric to control them. This will ease their coordination and it can even simplify the hardware with respect to the one required for implementing them separately, thanks to the reuse of the hardware that computes the metric. A metric per set like the one that DSBC requires cannot be obtained from the global counter used by DIP. Thus we checked whether the decision that DIP takes based on the set dueling can be made instead based on the SSL provided by the DSBC. This would not only simplify the design of the cache, but also avoid having always a fraction of the cache sets working with a wrong policy, even if this fraction is small. A way to achieve this is to use the SSL of

each set to decide whether the traditional insertion policy or BIP is better suited for that specific set. Our proposal is to change the insertion policy of a set to BIP only if it gets saturated (SSL=2K-1 for a saturation counter in the range 0 to 2K-1), and revert to MRU insertion when it reduces clearly its SSL. An SSL below K has been chosen to trigger the change to MRU insertion. This proposal, which we call Local DIP, only involves a saturation counter and one additional bit per set, called *insertion policy bit*, that indicates the insertion policy of the set. Local DIP is compared with DIP in terms of IPC improvement and miss rate reduction over a baseline 8-way second level cache of 2MB with lines of 64 bytes in Figures 3.1 (a) and 3.1 (b), respectively. BIP uses $\varepsilon = 1/32$ as the probability a new line is inserted in the MRU position of the recency stack instead of in the LRU one in both implementations. The simulation environment and the benchmarks are explained in Section 3.4. The results are similar, Local DIP being slightly better than DIP on average. Thus we dropped set dueling in favor of a local per-set decision based on its SSL.

Let us consider now the nature of the SSL. A high SSL indicates that the set cannot hold its working set, but it is difficult to know only with this value whether this is a problem specific to the set, which means other sets have no problems with their working sets, or a global problem of capacity of the cache. The answer lies in the comparison with the SSL of the other cache sets. If the cache has enough capacity to hold its working set, the DSBC mechanism should be able to find suitable sets to be associated to the problematic one, allowing it to displace part of the lines of its working set to a destination set with underutilized lines. If the DSBC cannot associate the set, that is because there are no sets with a SSL low enough to deem them good candidates to receive lines from other sets. This then points to a potential problem of capacity of the cache, which can be dealt with adopting BIP. Since DSBC only seeks to initiate associations when a set is saturated, a good strategy is to first try to associate the set to a destination set, and if no good candidate is found, change the set insertion policy to BIP.

Altogether this strategy equates to first trying to consider the high SSL in the set as a local problem, that is, conflict misses due to the oversubscription of this specific set, and if this fails, consider that there may be a global problem of capacity which requires turning to BIP. If the cache has a capacity problem, it is very likely that sets

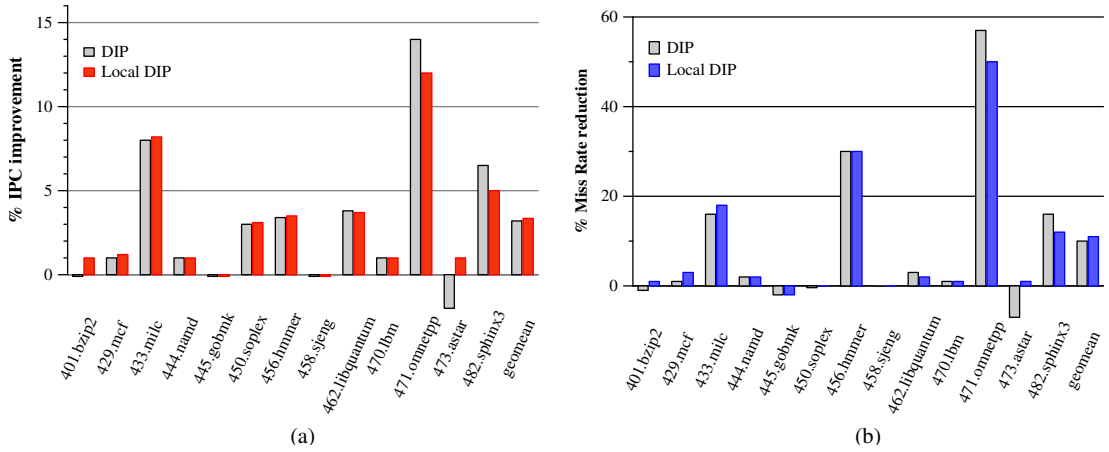


Figure 3.1: Percentage of IPC improvement (a) and miss rate reduction (b) related to a 2MB 8-ways and 64 bytes line size baseline cache using DIP and Local DIP.

that were chosen previously as destinations of an association become saturated too. Thus we propose that destination sets that become saturated change their insertion policy to BIP. Relatedly, it is logical that if the source set of an association gets saturated and its destination set is applying BIP, the source set changes to BIP too. This acknowledges that a capacity problem rather than a local conflict problem is being faced. Finally, just as in our Local DIP, if the SSL of a set in BIP mode drops below K, its insertion policy changes to MRU insertion, since the capacity problems seem to have disappeared.

The eviction of recently inserted lines in destination sets that operate under BIP by lines displaced from their source set was identified as one of the problems of the DSBC+DIP approach in Section 3.2. BIP puts useful lines in destination sets in a dangerous situation because, since they are inserted in the LRU position, any displacement before their reuse evicts them from the cache. Our design avoids this enforcing that destination sets in BIP mode are in read-only mode. This means that misses in their source set will lead to searches in them, but no displacements of lines from the source set will be allowed. This is consistent with our view that BIP is triggered to deal with capacity problems rather than conflicts. Another positive side-effect of this policy is that since no displacements are allowed in BIP mode, it is easier to break the association, which is in fact not helpful when there is a capacity problem. Let us recall that the association is broken when during its operation the destination sets evicts all the lines it received from the source set. Finally, when the

SSL of a destination set goes below K, besides reverting to MRU insertion, it also enables again the displacement of lines from its source set.

Altogether, our proposal, called Bimodal Set Balancing Cache (BSBC) because it is a Set Balancing Cache with an integrated BIP, has almost the same hardware overhead as a DSBC. Only one additional bit is required per set in order to store its current insertion policy. As for the time required to apply the BSBC algorithms, just as in the DSBC and DIP, they are triggered by misses and can be thus overlapped with their resolution. The contention in the tag array due to second searches has been considered in our evaluation.

3.4. Simulation environment

We have used the same simulation environment and executed the benchmarks under the same conditions as in Chapter 2. Still, we have increased the number of benchmarks tested in order to strengthen the obtained results. These additional benchmarks (*429.mcf*, *470.lbm* and *482.sphinx3*) are very memory-demanding, which results in big working sets and therefore high capacity miss rates. For this reason, the SBC can only help partially to improve their performance. This way in this evaluation 13 benchmarks from the SPEC CPU 2006 suite will be used. They are characterized in Table 3.1, which shows the number of accesses to the L2, the miss rate in the L2 cache both in the two-level (2MB L2) and the three-level (256kB) configurations, and the component of the suite they belong to.

3.5. Experimental evaluation

The BSBC and the compared techniques have been applied in the second level for the two-level configuration and in the two lower levels for the three-level configuration. As for the parameters that are specific to the different approaches used in this study, DIP uses 32 sets dedicated to each policy to decide between BIP and

Table 3.1: Benchmarks characterization. MR stands for miss rate.

Bench	L2 Accesses	2MB L2 MR	256kB L2 MR	Comp.
bzip2	125M	9%	41%	INT
mcf	720M	31%	56%	INT
milc	255M	71%	75%	FP
namd	63M	2%	5%	FP
gobmk	77M	5%	10%	INT
soplex	105M	8%	15%	FP
hmmer	55M	10%	41%	INT
sjeng	32M	26%	27%	INT
libquantum	156M	74%	74%	INT
lbm	580M	31%	32%	FP
omnetpp	100M	28%	91%	INT
astar	192M	23%	48%	INT
sphinx3	122M	68%	76%	FP

MRU insertion. This BIP as well as the one triggered by the BSBC use a probability $\varepsilon = 1/32$ that a new line is inserted in the MRU position of the recency stack. The DSBC and the BSBC use a Destination Set Selector of four entries (see Section 2.4.1).

Figures 3.2 and 3.3 show the performance improvement in terms of instructions per cycle (IPC) for each benchmark in the two-level and the three-level configurations tested, respectively. The figures compare the baseline not only with DSBC and DIP, but also with a combination of both approaches without coordination. This latter configuration is tested to show the importance of coordinating the behavior of policies designed to deal with different problems. The bar labeled *geomean* is the geometric mean of the individual IPC improvements seen by each benchmark.

In the two-level configuration the BSBC always has a positive or, at worst, in the case of 445.gobmk benchmark, a negligible negative effect on performance smaller than 1%. The geometric mean of the relative IPC improvement for BSBC with respect to the baseline configuration is 4.8% in the two-level configuration. DIP, DSBC and DIP+DSBC achieve 3.2%, 3.6% and 3%, respectively. The analysis based on IPC points again to the importance of the contributions of this Thesis. A non coordinated attempt to deal with capacity and conflict misses such as DIP+DSBC is outperformed by all the configurations tried.Nevertheless, our coordinated effort of DIP and DSBC guided by the SSL achieves clearly the best overall results. The

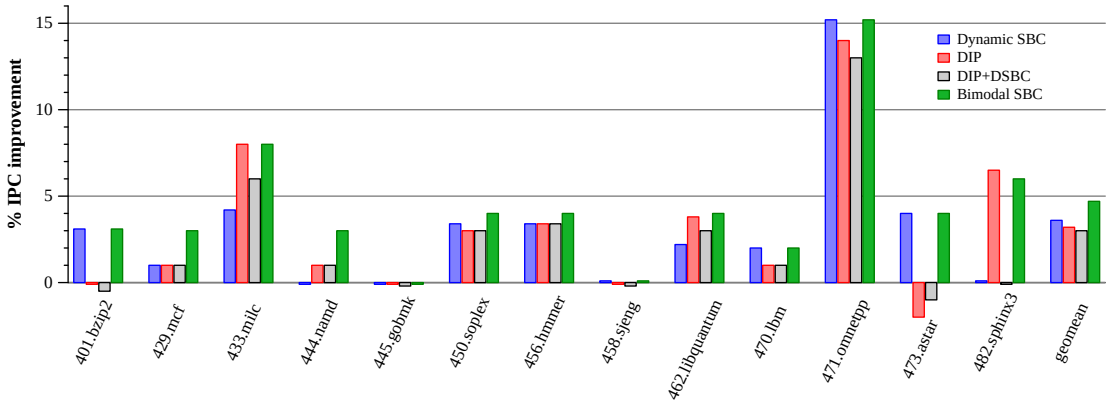


Figure 3.2: Percentage of IPC improvement over the baseline in the two-level configuration.

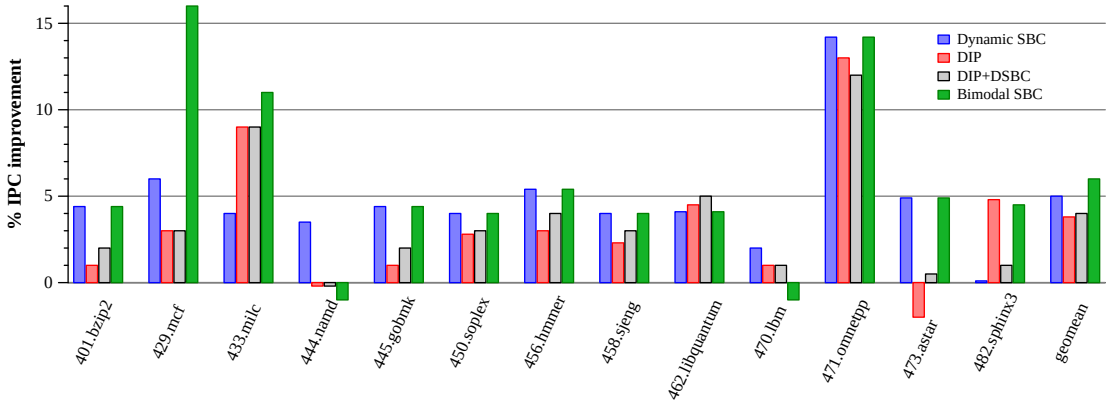


Figure 3.3: Percentage of IPC improvement over the baseline in the three-level configuration.

situation is very similar in the configuration with three levels of cache. Here BSBC improves 6% on average the IPC with respect to the baseline system, while DIP, DSBC and DIP+DSBC achieve increases of 3.8%, 5% and 4%, respectively.

Finally, BSBC is able to clearly outperform both static and dynamic versions in streaming applications like 429.mcf, 433.milc and 482.sphinx3 and only diminishing performance in 444.namd and 470.lbm. In these latter cases where applying a different insertion policy with little accuracy can be harmful, the ε parameter of the BIP insertion policy allows to better adjust its operation. We performed experiments applying different values, achieving 5.5% improvement using an $\varepsilon = 1/16$ and 5.2% with an $\varepsilon = 1/8$ over the three-level baseline configuration without degrading the performance of any benchmark.

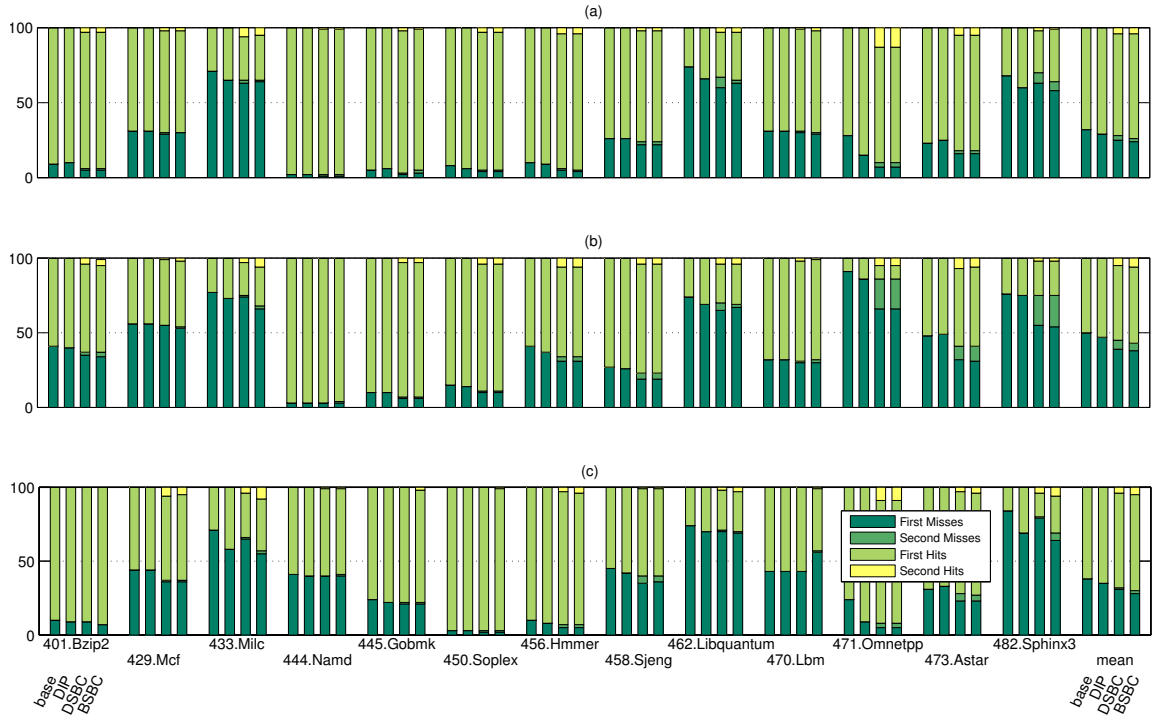


Figure 3.4: Miss, secondary miss, hit and secondary hit rates for the (a) L2 cache in the two-level configuration, (b) L2 cache in the three-level configuration, and (c) L3 cache in the three-level configuration.

The behavior of DIP, DSBC and BSBC is compared in the second level cache of the hierarchy with two levels of caches and the two lower levels of the hierarchy with three levels in Figure 3.4. It shows the rate of accesses that result in misses after a single access to the tag-array (primary misses) or two (secondary misses), and accesses that hit in the cache in the first check of the tag-array (primary hits) or after the second one (secondary hits). Only DSBC and BSBC present secondary accesses, which take place when an access misses in the source set of an association. The last group of columns (mean), represents the arithmetic mean of the rates observed in each cache. For example in the L2 of the two-level configuration the BSBC gets an average miss rate (considering both kinds of misses) of 25.2% compared to the 30% of the baseline configuration. This means a relative reduction in the miss rate of 16%. In this cache DIP achieves a miss rate reduction of 10% and DSBC of 11.5%. So we can see that our design allows the two policies to work coordinately getting

the best of each one of them. The ratio of all the cache accesses that result in secondary misses is 2% and 3% for BSBC and DSBC, respectively. The additional delay of a secondary miss with respect to a primary miss is small, but still it is good that BSBC not only generates more hits than DSBC but also reduces by 1/3 the number of secondary misses. The reduction is not surprising if we realize that in applications with capacity problems, the saturation of the destination sets will avoid displacements of lines that are actually not useful. This is will also enable these sets to break the association before. Let us remember that an association is broken when the destination set evicts all the lines received from the source set. Altogether this leads to fewer unsuccessful secondary searches than in DSBC. DSBC and BSBC present the same rate of accesses that result in secondary hits, about 4%.

Looking at individual benchmarks we can appreciate how BSBC adapts to the different types of applications, often performing better than both DIP and DSBC. For example, in 433.milc, 462.libquantum and 482.sphinx3, which are more suited to DIP than to DSBC, BSBC achieves similar results to DIP (somewhat worse in 482.sphinx3), and better than DSBC. The BSBC is also able to adapt to those applications that benefit more from DSBC because of imbalances in the working sets sizes for different cache sets. This happens, for example, in 401.bzip2, 471.omnetpp and 473.astar, where BSBC and DSBC work better than DIP. Therefore, the BSBC works largely as DIP for streaming applications using BIP, and mostly as the DSBC when the application presents imbalances among the working sets of sets. It is often the case that the BSBC even improves over both approaches by combining both behaviors.

From this figure we can see how BSBC is able to work largely like DSBC when there are unbalances among sets and largely like DIP when capacity problems appear in the cache. The coordination of both approaches is responsible for the large benefits of BSBC.

After evaluating BSBC related to its underlying techniques we analyze here how it performs in comparison with some related proposals. Figures 3.5 (a) and (b) compare the IPC improvement in the two and three-level configurations, respectively, among pseudoLIFO, DRRIP and the BSBC. DRRIP (using *Hit Priority*) [28] use 32 sets dedicated to each policy for each core to decide between BRRIP and SRRIP. This bimodal policy uses a probability $\varepsilon = 1/32$ that a new line is inserted with a

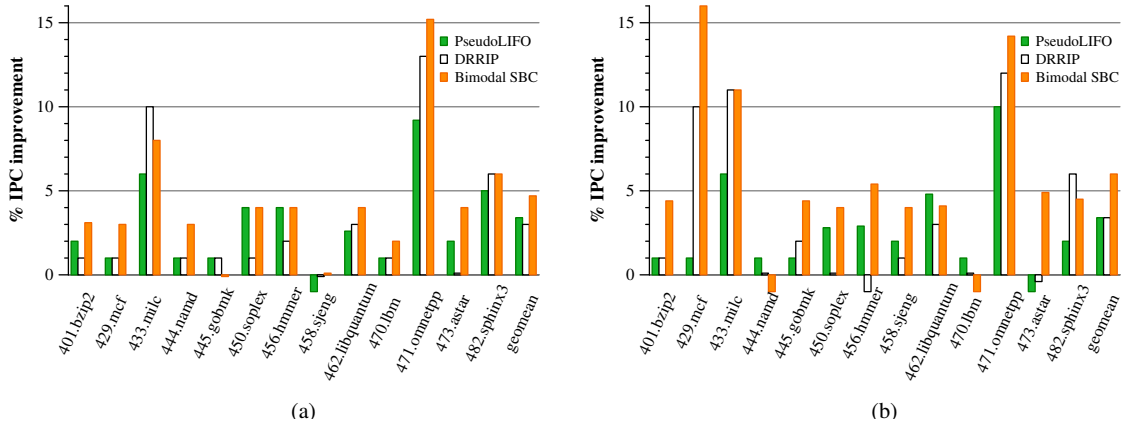


Figure 3.5: Comparison with recent proposals in terms of IPC improvement in the two (a) and three-level (b) configurations.

long re-reference interval prediction. The probabilistic escape LIFO we evaluate, or pseudoLIFO in what follows, approximates the hit counts by the next power of two for escape probabilities and uses 4 dedicated sets every 1024 sets in the cache for each one of the four policies described in [9]. The last group of bars corresponds to the geometric mean.

In Figure 3.5 (a) we can observe how BSBC achieves the best performance in terms of IPC, followed by pseudoLIFO with 3.4% and, finally, DRRIP 3%. As for the three-level configuration, Figure 3.5 (b) shows that pseudoLIFO and DRRIP achieve the same degree of IPC improvement: 3.4%. We can observe that pseudoLIFO outperformed DIP and DIP+DSBC in the two-level configuration (see Figure 3.2). Nevertheless the opposite happened in the three-level configuration (compare Figure 3.5 (b) with Figure 3.3), and by somewhat larger margins. This, coupled with the negligible hardware cost and simple algorithm of DIP were the main reasons to choose BIP as the preferred approach to deal with capacity misses in the BSBC design.

As for the miss rate reduction, we can see in Figure 3.6 that the results vary between the 16% reduction for the BSBC and the 11% of the pseudoLIFO and DRRIP approaches. Benchmark by benchmark, DRRIP is the best one in three of them while BSBC in nine. We must take into account that most of the approaches turn misses into hits, while the BSBC turns them into secondary hits, which suffer the delay of a second access to the tag array.

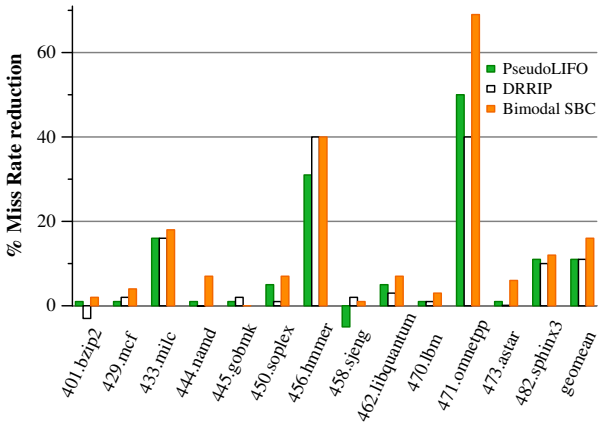


Figure 3.6: Miss rate reduction of the BSBC and some recent proposals in the two-level configuration.

3.5.1. Average memory latency and power consumption

Figures 3.7 and 3.8 show the percentage of power consumption and latency reduction achieved by BSBC related to the two and three-level baseline configurations, respectively. The bars for both magnitudes are broken down in the percentage of hits that are satisfied in the second (and third level for the latter case) level of the memory hierarchy or in the main memory. We do not show the percentage due to the L1 hits because it is quite similar for all the approaches. Using CACTI [21] we have deduced the power consumption due to second searches. As the tag check delay means only a 3% of the total power consumption per read/write access in the cache, our approach has a negligible power consumption overhead. Note that the percentage that second hits represent in terms of power consumption and latency are considered in this analysis.

3.6. Cost

We consider here the cost of BSBC in terms of storage. The cost of DSBC, already discussed in Section 2.7, is also computed for comparison purposes, the cost of DIP being negligible (just 10 bits for the global counter).

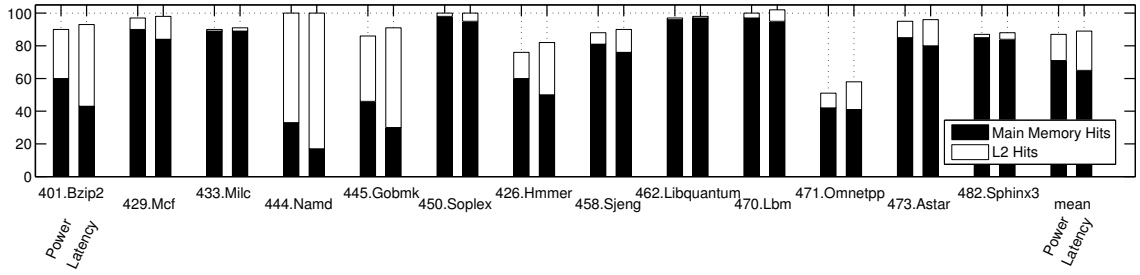


Figure 3.7: Average power consumption and memory latency reduction achieved by the BSBC and breakdown of the accesses in the **two-level** configuration related to the baseline.

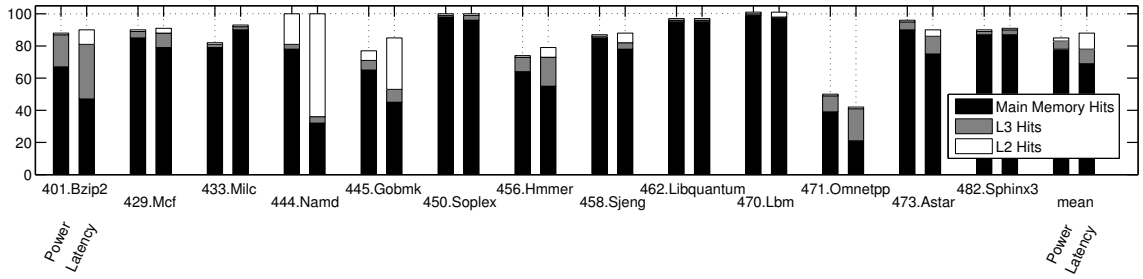


Figure 3.8: Average power consumption and memory latency reduction achieved by the BSBC and breakdown of the accesses in the **three-level** configuration related to the baseline.

The BSBC, like the DSBC, requires the following additional hardware with respect to a standard cache: a saturation counter per set to compute the SSL, an additional bit per entry in the tag-array to identify displaced lines (d bit), an Association Table with one entry per set that stores a bit to specify whether the set is the source or the destination of the association, and the index of the set it is associated to, and finally a Destination Set Selector (DSS) to choose the best set for an association. A 4-entry DSS has been used in our evaluation. The BSBC needs also one bit per set to indicate the set insertion policy. Based on this, Table 3.2 calculates the storage required for a baseline 8-way 2 MB cache with lines of 64B assuming addresses of 42 bits.

Table 3.2: Baseline, DSBC and BSBC storage cost in a 2MB/8-way/64B/LRU cache

	Baseline	DSBC	BSBC
Tag-store entry:			
State(v+dirty+LRU+[d])	5 bits	6 bits	6 bits
Tag ($42 - \log_2 sets - \log_2 64$)	24 bits	24 bits	24 bits
Size of tag-store entry	29 bits	30 bits	30 bits
Data-store entry:			
Set size	64*8*8 bits	64*8*8 bits	64*8*8 bits
Additional structs per set:			
Saturation Counters	-	4 bits	4 bits
Insertion policy bit	-	-	1 bit
Association Table	-	12+1 bits	12+1 bits
Total of structs per set	-	17 bits	18 bits
DSS (entries+registers)	-	4*(1+12+4)+2*(2+4) bits	4*(1+12+4)+2*(2+4) bits
Number of tag-store entries	32768	32768	32768
Number of data-store entries	32768	32768	32768
Number of sets	4096	4096	4096
Size of the tag-store	116kB	120kB	120kB
Size of the data-store	2MB	2MB	2MB
Size of additional structs	-	8714B	9226B
Total	**2164kB**	**2176kB** (0.55%)	**2177kB** (0.6%)

3.7. Analysis

In this section we evaluate how the performance and cost of BSBC vary with respect to the parameters of the cache. Also we have analyzed its internal behavior. Finally, we study the BSBC performance in a multicore environment.

3.7.1. Impact of varying cache parameters

All along this section we will always use as baseline the 2MB/8-way/64B/LRU L2 cache of our two-level configuration. Table 3.3 shows the miss rate reduction achieved by BSBC as well as the storage overhead it involves as the cache size varies between 256kB and 4MB. The BSBC always reduces the average miss rate obtained, but as the cache size increases the working set of some benchmarks fits better, reducing the opportunities of improving it.

Table 3.3: Cost-benefit analysis of BSBC as a function of the cache size.

Cache size	Baseline miss rate	BSBC miss rate	BSBC miss rate reduction	BSBC storage overhead
256KB	48.4%	42%	13.2%	0.53%
512KB	43.1%	36%	16.4%	0.55%
1MB	36.5%	30.8%	15.6%	0.57%
2MB	30%	25.2%	16%	0.6%
4MB	24%	20.5%	14.5%	0.63%

Table 3.4: Cost-benefit analysis of BSBC as a function of the line size.

Line size	Baseline miss rate	BSBC miss rate	BSBC miss rate reduction	BSBC storage overhead
64B	30%	25.2%	16%	0.6%
128B	27.4%	23.4%	15.6%	0.28%
256B	24%	20%	16.6%	0.13%

Table 3.5: Cost-benefit analysis of BSBC as a function of the associativity.

Associativity	Baseline miss rate	BSBC miss rate	BSBC miss rate reduction	BSBC storage overhead
8-ways	30%	25.2%	16%	0.6%
16-ways	29.3%	24.6%	16%	0.37%
32-ways	29%	24.3%	15.2%	0.28%

Table 3.4 studies the cost-benefit of BSBC comparing the miss rate reduction achieved by it versus the additional storage cost it incurs as a function of the line size in the baseline cache. The increase in the line size reduces proportionally both the number of lines and sets, being the BSBC cost mostly proportional to the latter as we can see.

Table 3.5 makes the same study from the point of view of the associativity considering values of 8, 16 and 32. The increase of associativity reduces the number of sets, and thus the relative cost of BSBC, but increases the tag size. Miss rates and their reduction stay very flat. Also, just as the experiments in Section 3.5 considering caches that duplicated the associativity, this table shows that making shared usage of the lines of two sets under heuristics like the ones proposed by BSBC is much more effective than organizing the cache lines in sets with twice or even four times larger.

3.7.2. BSBC behavior

Firstly, we analyze here the percentage of misses that occur in the BSBC which insert the new line in the MRU or in the LRU position, that is, applying the traditional insertion policy or BIP, in comparison with DIP. Then, we analyze the BSBC

behavior based on measurements in the L2 cache of the two-level configuration. In this latter case, we have performed some experiments on particular parameters of DSBC, a combination of DSBC and DIP without coordination (DSBC+DIP) and, finally, BSBC. Namely, we gathered the percentage of lines displaced that are never used, the average number of reuses per displaced line and the number of lines evicted from a set which are the next miss in that set. This latter value is the number of times that a given line X is evicted from a set to insert a new line, and then (after may be several hits in the middle), the next miss happens on that line X.

Figure 3.9 shows the miss rate reduction for both DIP and BSBC related to the baseline, which is represented by the horizontal dotted line in 100. The ratio of misses is broken down into the percentage of misses which insert the new line in the MRU position, applying the traditional MRU insertion policy, or in the LRU position, applying BIP. Note that, in this case, we are referring to the original set, because second searches which result in misses do not imply a new insertion in the destination set. The average number of insertions which take place in the LRU position is 48%, 78% and 28% in the L2 cache of the two-level configuration, in the L2 of the three-level configuration and, finally, in the L3 of the three-level configuration, respectively. As for DIP, it inserts lines in the LRU position with a probability of 54%, 82% and 60%, respectively.

As for the average number of hits per displaced line, DSBC achieved a 3.3 on average, DSBC+DIP 1.1 and, finally, BSBC 6. It is worthy to point out that the number of displacements for DSBC and BSBC is similar, unlike DSBC+DIP, that performs about 20% more, achieving a lower ratio of hits per displaced line as well. Regarding the average number of displaced lines that are never used, its percentage is 28% for DSBC, 37% for DSBC+DIP and 24% for BSBC. Finally, the percentage of misses that go to the last evicted line in the set is, on average, 1% for DSBC, 3% for DSBC+DIP and 0.5% for BSBC.

We can conclude, according to these results, that BSBC is able to reduce the harmful behaviors that may appear if DSBC and DIP are combined without coordination. Also, we can infer that our designs select in a more accurate way which lines should be displaced as well. However, the prevention of evicting lines that have been very recently inserted in the cache and which presumably have little locality is not the main reason for the difference between the performance obtained by BSBC

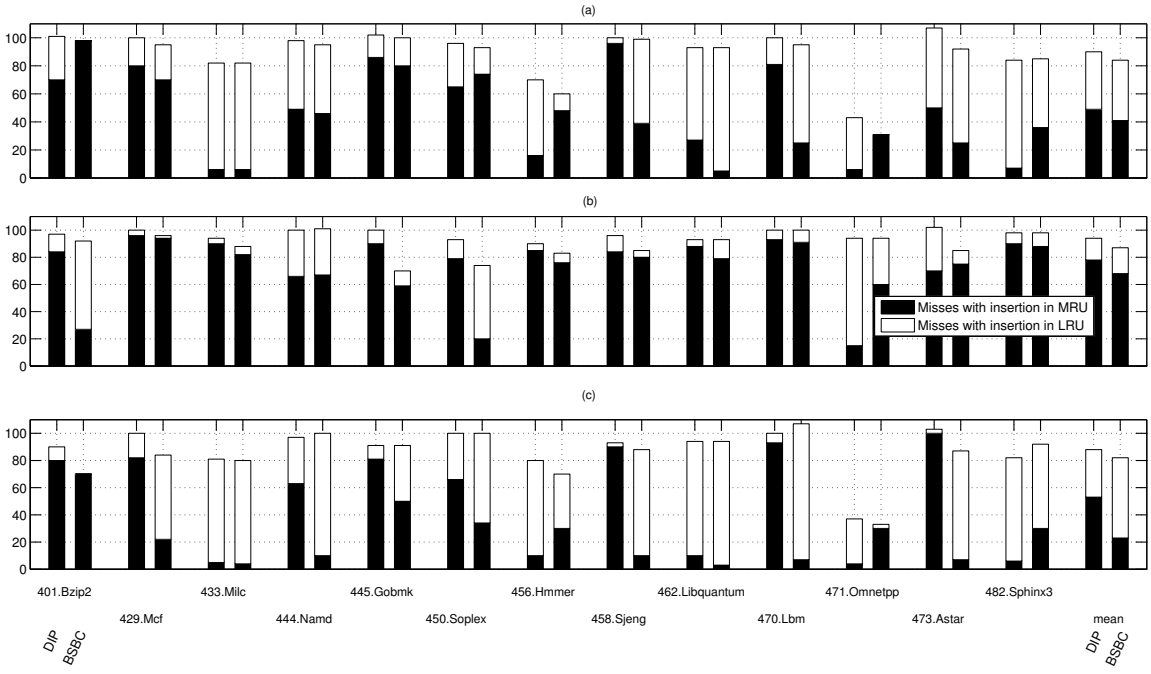


Figure 3.9: Insertions in the MRU and LRU position after a miss in the DIP and BSBC designs for the (a) L2 cache in the two-level configuration, (b) L2 cache in the three-level configuration, and (c) L3 cache in the three-level configuration.

related to the other approaches. We believe that the possibility of choosing the insertion policy at the cache set level, rather than for the whole cache at once, like in other approaches, is more important. Moreover, avoiding the displacement of useless lines to destination sets when they are saturated brings great improvements for BSBC, since it gets a ratio of hits per displacement high enough to make the percentage of useless displaced lines less critical.

3.7.3. Multicore experiments

The concepts underlying the Set Balancing Cache are directly applicable to a multicore environment. We have performed experiments using the same two-level baseline system described in Table 2.1 with a shared L2 cache instead. The bench-

Table 3.6: Multiprogrammed workloads characterization.

Name	Benchmarks	L2 Miss rate	MPKI
MW1	471.omnetpp + 473.astar	42%	22.7
MW2	433.milc + 482.sphinx3	70%	28.5
MW3	401.bzip2 + 462.libquantum	39.5%	14.8
MW4	462.libquantum + 470.lbm	36.1%	45.9
MW5	401.bzip2 + 429.mcf	26.7%	35.9
MW6	429.mcf + 433.milc	61%	103
MW7	444.namd + 471.omnetpp	24%	7.1
MW8	456.hmmer + 482.sphinx3	55.5%	11.1
MW9	445.gobmk + 473.astar	15%	5.5
MW10	462.libquantum + 471.omnetpp	70%	28.6
MW11	433.milc + 473.astar	48.8%	22.7
MW12	458.sjeng + 482.sphinx3	55.4%	9.6
MW13	429.mcf + 444.namd	18.4%	19.6
MW14	429.mcf + 445.gobmk	22.7%	27.6
MW15	433.milc + 470.lbm	37.8%	56.7
MW16	401.bzip2 + 458.sjeng	10.8%	1.8

marks described in Table 3.1 have been combined in order to make 16 multiprogrammed workloads of two applications and 6 of four with at least an MPKI of 1 (details are shown in Table 3.6 for the former and in brackets in each graph for the latter). They have been simulated using the same parameters as in Section 3.5. When each core commits the preestablished number of instructions (10^{10}) it continues its execution until the last core finishes, in order to keep competing for the shared resources in the L2 cache.

Our evaluation relies on metrics usually referenced in the bibliography: throughput, weighted speedup [67], which indicates execution time reductions, and the harmonic mean of weighted speedups [48], which balances fairness and performance. We have compared our designs with some recent techniques which were briefly described in Section1.2.2.1. The Thread-Aware Dynamic Insertion Policy (TADIP) [27] has been simulated (specifically TADIP-Feedback) using 32 sets dedicated to each policy for each core to decide between BIP and MRU insertion. BIP uses a probability $\varepsilon = 1/32$ that a new line is inserted in the MRU position of the recency stack. The Pseudo-LIFO policy used in our experiments, which proved to benefit shared caches [9], approximates the hit counts by the next power of two for escape probabilities and uses 4 dedicated sets every 1024 sets in the cache for its particular policies. Also, we have included PIPP [84] in our tests, using 32 dedicated sets to track the

hit counters information in the utility monitors and a probability of promotion of 3/4 and 1/128 in normal and streaming mode, respectively. The cache switches to streaming mode if the number of misses per interval is greater than 4095 or the miss rate in the shadow tags exceeds $12,5\%$.

Figure 3.10 shows the percentage of throughput improvement over the baseline for the different approaches considered, running 2 applications. We can see how the DSBC and BSBC average improvements, 2% and 3% respectively, are smaller than those reported in the single core platform. Despite that, both approaches achieve improvements of up to 10%, only PIPP gets a better peak improvement. BSBC obtains the best overall results. Second comes TADIP thanks to its thread-aware design, followed by DSBC.

Despite the moderate throughput improvement DSBC and BSBC get, both techniques achieve a high miss rate reduction in the shared L2 cache, being the L1 cache miss rate almost the same in all approaches. Figure 3.11 shows that this reduction is 7.8% and 10% related to the baseline, respectively. TADIP gets a 8%, PseudoLIFO 7% and, finally, PIPP 5%.

The studies for the other common metrics related to speedup show very similar trends to those observed in Figure 3.10 for the throughput improvement. Therefore, Figure 3.12 (a) and 3.12 (b), devoted to the weighted speedup and the harmonic mean of weighted speedups or fairness, respectively, only show the values for BSBC and TADIP, the two approaches which performed better in the throughput experiments. BSBC outperforms the baseline configuration by 2.2% and 3.4% in terms of weighted speedup and fairness, respectively, while TADIP does it by 2.2% and 3.3%. In order to see how DSBC and BSBC work when the number of cores increases, Figure 3.13 shows the same study as in Figure 3.10 but running 4 applications that share a 4MB 16-ways L2 cache. This time, DSBC is outperformed by all approaches but PseudoLIFO. As for BSBC, although it is only outperformed by TADIP, it achieves a small 2% of throughput improvement over the baseline, like PIPP. We can see how approaches specifically designed for multicore environments, like TADIP, get better results as the number of cores increases due to its strength and thread-awareness.

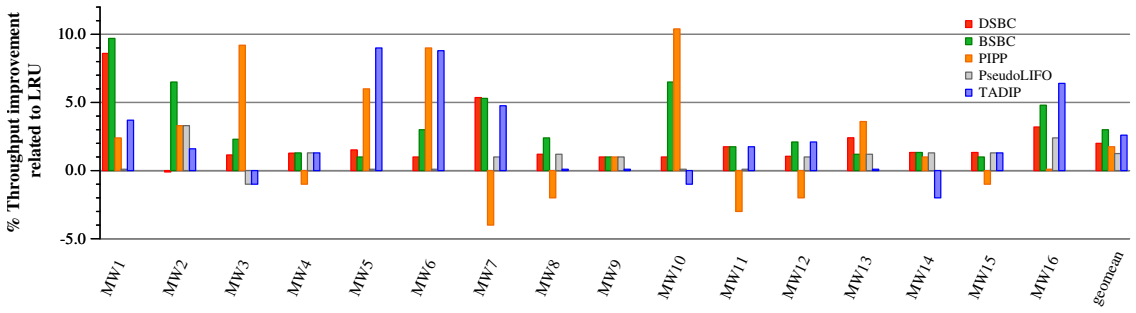


Figure 3.10: Percentage of throughput improvement over the baseline configuration using several policies when running **2** applications.

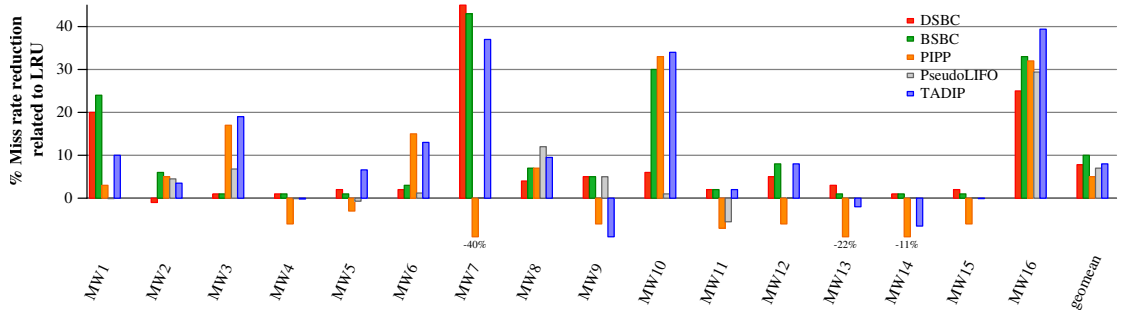


Figure 3.11: Percentage of miss rate reduction over the baseline configuration using several policies when running **2** applications.

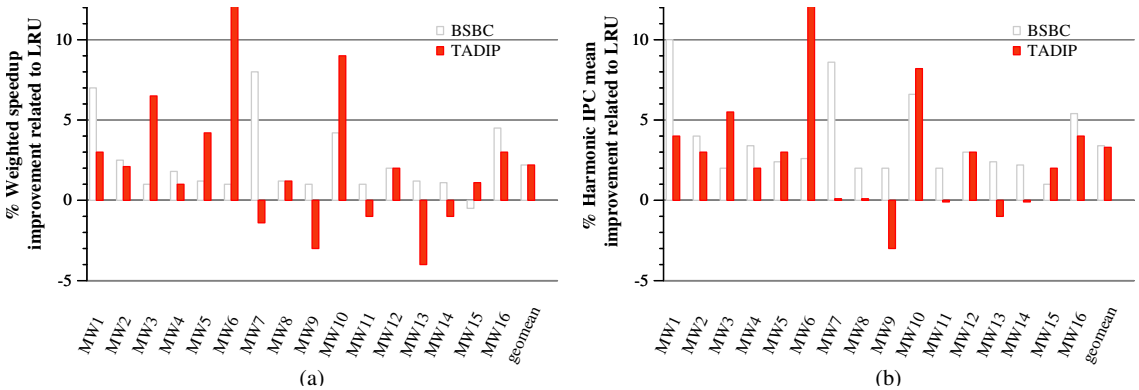


Figure 3.12: Percentage of weighted speedup (a) and harmonic IPC (fairness) improvement (b) over the baseline configuration using BSBC and TADIP.

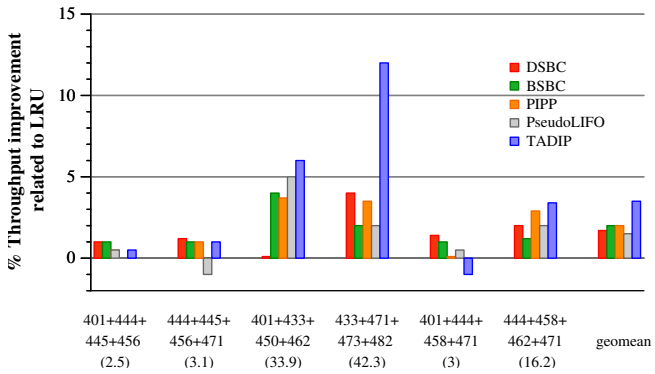


Figure 3.13: Percentage of throughput improvement over the baseline configuration using several policies when running **4** applications.

3.8. Summary

There has been extensive research to improve the behavior of caches, particularly of non first-level ones. Different approaches are sometimes better suited to reduce different kinds of misses. For example, DSBC [60] and DIP [53] target somewhat different problems. If implemented jointly in the cache, complementary techniques like these ones should have the potential to achieve better performance than any of them isolated. Nevertheless, as this chapter shows with the case of DSBC and DIP, implementing them at the same time is not enough to exploit their advantages. In fact, the direct simultaneous application of these techniques often yields worse results than the usage of only one of them. This chapter introduces a coordinated mechanism that combines both techniques in order to increase performance by tackling conflict and capacity misses at the same time. We have analyzed the reasons for this behavior and proposed in a reasoned way an integrated design of these policies that allows them to cooperate effectively. As part of this design the usefulness of the Set Saturation Level (SSL) metric to detect problems of capacity as well in the cache is demonstrated.

Simulations using benchmarks with varying characteristics show that, when properly integrated with the proposed Bimodal Set Balancing Cache (BSBC) design, the joint application of the DSBC and BIP policies goes from being often one of the worst approaches to being the best one. For example in a 2MB, 8-way second level cache DIP+DSBC jointly reduces the miss rate by 8.3% in relative terms, while

DSBC and DIP reduce it by 12% and 10% respectively. With BSBC the relative miss rate reduction almost doubles to 16%. This leads also the BSBC to get the largest IPC improvement, 4.8% on average for this configuration, compared to the 3% that a straight DSBC+DIP implementation provides.

Additionally, we have also shown that BSBC is directly applicable to shared caches, where it has performed well, achieving 10% of miss rate reduction on average and 3% of throughput improvement. This is even better than some techniques specifically designed to work on this kind of platforms, like PIPP [84]. The DSBC has been also evaluated in shared caches, reducing the baseline miss rate by 7.8% on average and achieving throughput improvements of up to 10%. Despite that, techniques that include support for thread-awareness, like TADIP [27], have performed better in our experiments considering a shared LLC when the number of cores increases. This calls for the extension of BSBC with thread-awareness to achieve better results in shared caches.

Chapter 4

Virtually Split Cache

4.1. Introduction

In the previous chapter we concluded that different access streams, specifically those generated by different threads sharing cache resources, should be handled by applying particular treatments to each one of them. Extending this concept to first-level caches would mean to separately consider the two main streams these caches deal with, namely those due to data and instruction accesses. First-level caches with a split design for instructions and data are commonly found nowadays in modern processors. This design is the preferred one, rather than a unified approach, as it is well suited for the design of the processor pipeline so that different stages access different caches, there being no conflicts between them and achieving a better memory bandwidth than a unified cache. Also, the instruction cache design is simplified as it only requires to support read operations, while a unified or data cache needs hardware to deal with both read and write operations. However, unified approaches, usually found in the lower levels of the memory hierarchy, provide a better use of resources by automatically sharing the cache capacity for both instructions and data, at the expense of requiring a higher latency, design complexity and limiting the total bandwidth. Also, a first-level unified cache would usually need to be multi-ported; so that it could process simultaneously both instructions and data requests, which requires more complexity. Furthermore, the higher locality instructions usually have compared to data is not considered in unified approaches, which are thus unaware of

the different space requirements instructions and data could demand. We propose in this chapter the Virtually Split Cache or VSC, a middle-way solution aimed to embrace the advantages that both split and unified designs provide while it alleviates their drawbacks. This new design distributes cache resources between data and instructions depending on their particular demand. Finally, this new technique uses a bank level granularity in order to simplify its design, strongly restricted by the issue logic.

4.2. Background and Motivation

Nowadays, the most usual configuration for the first level of the cache memory hierarchy devotes independent caches for both instructions and data. This is the preferred configuration, mainly, because of the following reasons:

- Different instructions can access the instruction cache, in the fetch stage of the pipeline, and the data cache, usually in the memory stage, at the same time in pipelined processors. Unified approaches would require several access ports and, thus, more complexity to provide the same advantages.
- The instruction cache design may be simpler, as it only needs to perform, ideally, read operations.
- Unified caches of the same aggregated capacity imply higher latencies.

Nevertheless, unified approaches provide higher hit rates by automatically sharing resources instead of statically partitioning them.

Several designs have appeared in the last years in order to improve performance or reduce power consumption in first-level caches.

As for the first-level **data** caches, many techniques have been proposed in order to increase performance. Besides increasing associativity, logically or virtually as pseudo-associative caches do [2][6][87], or early evicting dead blocks in order to retain in the recency stack data with higher locality [43][46], other techniques oriented to better distribute the memory references across the cache in order to reduce conflict misses have also appeared. For instance, the B-Cache [86] tries to balance the

accesses to the sets of first-level direct-mapped caches by increasing the decoder length and incorporating programmable decoders and an *ad-hoc* replacement policy. Also, other approaches try to reduce cache access latency by partitioning the first-level data cache in order to place data near those units which are more likely to use them [56], leverage in software techniques to increase performance [47] or merge different designs to reduce power consumption [23].

Regarding the first-level **instruction** caches, techniques to increase performance have been focused on code layout optimization [76] or code reorganization [42] due to the different locality properties and access patterns instructions usually have compared to data.

These previous approaches concentrate their efforts on improving the design of only one kind of cache, but they are not aware of the underutilized space instruction or data caches may provide. This way, they are unable to balance resources in the first level of the memory hierarchy.

On the other hand, unified cache designs are usually found in lower levels of the memory hierarchy, where issue logic restrictions and latency constraints are more relaxed. Many approaches have been proposed in order to increase performance at these levels, both for private and shared configurations between several cores, due to the recent appearance of chip multiprocessors (CMPs). Recent approaches are mainly oriented to reduce capacity misses [53], conflict misses [60] or both [61][85]. There are also techniques specifically oriented to increase performance in shared last-level caches (LLCs) of CMPs, which usually apply partitioning mechanisms [13][54] to limit the amount of space devoted to each core.

Apparently, there is no previous research on unified caches focused on applying different policies for instructions and data, that is, there are no unified approaches that become aware of the different locality that instructions and data may have. Even more, the design of hybrid approaches with characteristics of shared and split caches has not ever been considered either.

In order to motivate the explanation of this possibility, we have performed studies increasing the cache size for both the instruction and data first-level caches in order to emphasize the different locality and space requirements both caches have. Figure 4.1 shows the evolution of the miss rate for both instruction and data caches

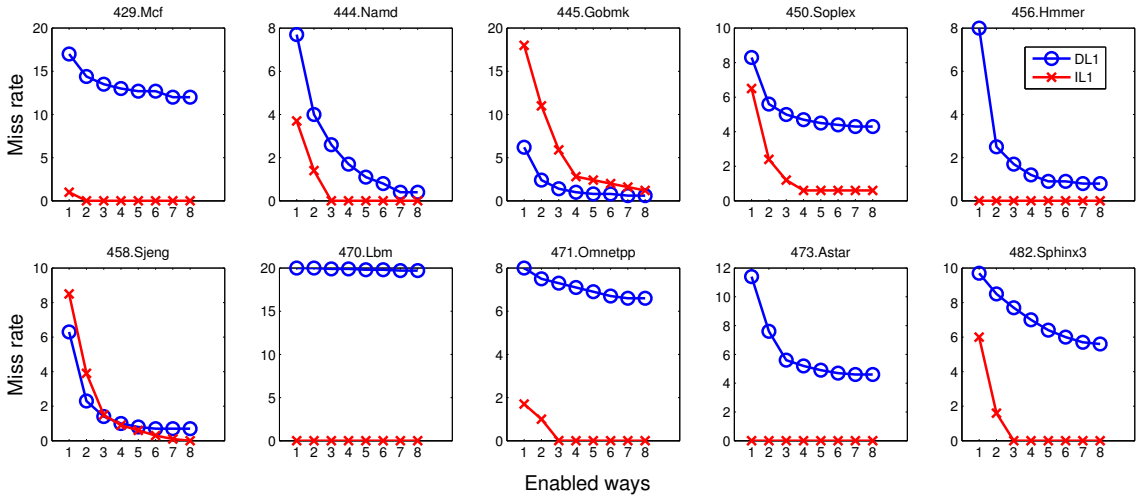


Figure 4.1: Miss rate for SPEC CPU2006 benchmarks as the number of allocated ways varies for both instruction and data first-level caches. The X axis shows the number of ways allocated from a 8-way 64KB cache (the remaining ways are disabled).

varying their size while the number of sets remains unchanged at executing 10 benchmarks from the SPEC CPU2006 suite. Further information about the evaluation parameters can be found in Section 4.5. Results go from a direct-mapped 8KB cache through a 8-ways 64KB cache, enabling one additional way in every cache related to the previous one. Our 32KB 4-way baseline instruction and data caches, which will be further described in Section 4.4, lay in between. Results show that for a first group of benchmarks such as *444.namd*, *471.omnetpp* or *482.sphinx3*, 3 ways are enough to achieve a miss rate close to that one obtained using 8 ways in the instruction cache, that is, allocating more than 3 ways in the instruction cache does not provide better performance. On the other hand, allocating more than 4 ways in the data cache usually means a lower miss rate for these benchmarks. This is due to the lower locality, in terms of both space and time, that data have related to instructions. Therefore, instruction caches can provide data caches with space in order to better balance the resources of the memory hierarchy for this group of benchmarks. A second group includes those benchmarks where the opposite behavior happens. For instance, during the execution of the *445.Gobmk* and *458.Sjeng* benchmarks, the instruction cache can benefit from receiving underutilized ways from the data cache

from enabling 4 ways on. Finally, a third group embraces those benchmarks, usually streaming applications, where both instruction and data caches are not hardly influenced by the number of enabled ways. An example of this behavior can be observed in benchmark *470.Lbm*.

4.3. Virtually Split Cache

We propose a new cache design, aimed for the first level of the cache memory hierarchy, which provides the benefits of a unified approach while not bringing its inconveniences. Our design can be described as a k-way associative cache whose data-store and tag-store are partitioned so that part of them will be devoted to caching instructions while the other part will cache data, giving place to two virtual caches. Each one of these caches has its own port(s), which operate independently, acting therefore as a traditional split first-level cache. Tag and data stores are often partitioned in banks in order to achieve power, latency and/or bandwidth improvements [33][69][73][78]. In our design both stores are partitioned in k banks each, one per cache way. This way, each bank has as many sets as the cache, but it holds the tag (in the tag-store) or the line (in the data-store) of a single way. When a traditional k-way associative cache with this design is accessed, the k banks of its tag-store are read in parallel so that their content for the selected set is compared with the requested tag. The k banks of the data-store are also read in parallel in the meantime in order to minimize the latency in case one of these comparisons results in a hit, as in that case the data from the corresponding bank will be immediately sent to the processor. Our proposal, called Virtually Split Cache or VSC, divides its k banks, or correspondingly ways, in each store in two groups, one for instructions and another one for data, which operate as two independent caches. The number of ways assigned to each cache, each way corresponding to the pair composed of one bank of the tag-store plus one bank of the data-store, is decided dynamically by our design based on an analysis of the behavior and space requirements of the instructions and data streams.

Our design needs some modifications with respect to a standard k-way cache in order to be able to dynamically assign different roles to its banks. Figure 4.2 shows a tag-store bank (or way) of the VSC. The line $i/\overline{d}_j$ indicates whether way j of the

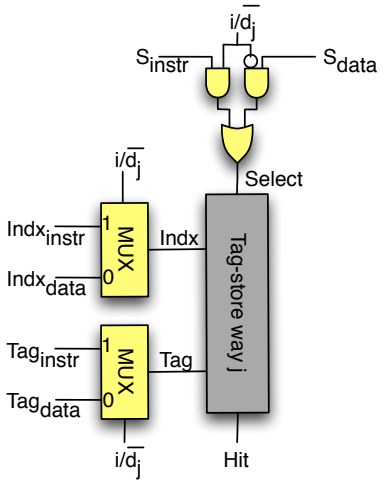


Figure 4.2: Way j in the tag-store of the Virtually Split Cache.

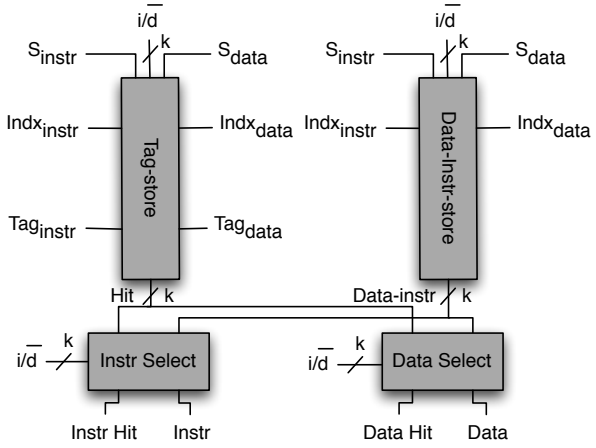


Figure 4.3: Virtually Split Cache general structure.

cache is assigned to instructions (value 1) or data (value 0). This signal controls the multiplexers that choose whether the address lines to select the set that must be indexed in the bank (*Indx*) and the tag that must be stored or compared (*Tag*) are those than come from the port for instructions or data. Similarly, it selects the appropriate selection line (S_{instr} or S_{data}) so that the bank will only perform accesses requested by the corresponding port. The modifications in the banks of the data-store of the VSC are analogous. Figure 4.3 shows the general structure of the VSC. Both the tag-store and the data-store, which is called data-instr-store in the figure to outline it can store both kinds of information, receive all the signals needed to operate each bank, both from the instruction and from the data port. They also receive the lines $i/\overline{d}_j, 0 \leq j < k$ so that each one of them controls whether the j-th bank of the store is assigned to either instructions or data, and operates therefore under the signals from that port. The selection boxes in the lower part of the figure take the k hit/miss lines from the tag-store, the values read from the data-instr-store, and the $i/\overline{d}$ lines that indicate whether each bank is assigned to data or instructions. With this information they can calculate in a straightforward way whether the current instruction (data) access has resulted in a hit or not, and in the first case, select the value from the associated bank and provide it to the processor.

We have assumed a single bank per way in each store and a single port per virtual cache in order to simplify the explanation. Nevertheless the banks of the VSC can be further subdivided as long as all the banks associated to the same way are controlled by the signals from the same virtual cache. This way, they could be subdivided to save energy [73], or to distribute the sets among interleaved subbanks in order to support multiple ports per virtual cache [59][69]. Any other of the usual strategies to implement multiple cache ports [59] could also be applied to the VSC. Additionally, while in the abstract representation in Figure 4.3 the tag-store and the data-instr-store have been separately depicted in order to simplify the representation, a smarter organization can be implemented. Namely, the tag-store banks could be interleaved with the data-instr-store banks, so that the two banks that form a way are nearby. In this situation, if the instructions port and the data port were in opposite sides of the array of banks, and the ways assigned to each virtual cache were always the nearest ones to its port, the wire delay of the VSC would be minimized. This smart design will be the one evaluated in Section 4.5.

There remains the issue of the algorithm to allocate ways to the virtual caches. Our design uses a counter I of ways that must be assigned to the instruction cache. All the lines $i/\overline{d}_j, 0 \leq j < I$ are set to 1, the ones for $I \leq j < k$ being set to 0. Our approach initially allocates half the ways for instructions and the other half for data. During its operation, the number of ways allocated to each virtual cache varies depending on each particular demand, provided that both caches have always one allocated bank at least. When a bank is reassigned to another virtual cache its contents are invalidated. This implies that this new allocated line per set is likely to be selected as a victim during the next replacement operation. We now explain in turn two practical designs of the VSC that use different algorithms to measure the demand of the virtual caches and decide the number of ways assigned to each one of them.

4.3.1. Shadow Tag VSC

This first approach tracks the space requirements for both instructions and data by estimating how well both streams would work if they acquired one additional way per set or, globally, an additional bank. Two shadow tags per set are used

in order to achieve this. One shadow tag stores the last instruction tag evicted from the set and the other one stores the last data tag evicted from the same set. Each time a miss occurs in a certain set the appropriate shadow tag, depending on the kind of request, is checked. If a hit occurs in the shadow tag and this is an instruction request, a global counter devoted to instructions is increased. Similarly, we use another global counter for data misses that hit in the data shadow tag as well. Also, as allocating an additional bank to a virtual cache implies deallocating it from the other cache, this approach needs additional structures to predict the performance loss implied in a virtual cache if it is deprived of one of its ways. For this purpose, our design uses two additional counters to track how many hits take place in the LRU position of every set for both instructions and data. Periodically, each 1 million cache accesses in our experiments, all 4 counters are checked. If the value of the shadow tag counter for data is greater than the value of the LRU counter for instructions, one bank initially devoted to instructions is allocated for data; and vice versa. The reason is that if the number of hits in shadow tags for data is higher than the number of hits in LRU positions for instructions, allocating one additional bank for data brings more benefits in terms of performance than keeping the same bank devoted to instructions. Analogous conclusions can be obtained for the opposite situation. Note that the number of allocated banks for instructions and data remains unchanged if both conditions or none of them are fulfilled. After this process the four counters are reset. Figure 4.4 shows the structure of this approach.

4.3.2. Global Selector VSC

While having a pair of shadow tags per set implies a small storage overhead (see Section 4.6), cheaper alternatives can be explored. This way we propose a lighter design which gets similar results using a common saturation counter, or global selector, for both instructions and data. This global selector is increased each time a miss occurs for an instruction request and decreased in case of a data miss. The counter value is checked after an update. If the global selector holds the maximum value, it means that instructions need more space. One additional bank, used for data hitherto, is allocated for instructions in this case. If the saturation counter has its minimum value, it means that more space for data must be provided. Our approach selects one instruction bank and allocates it for data. The global selector

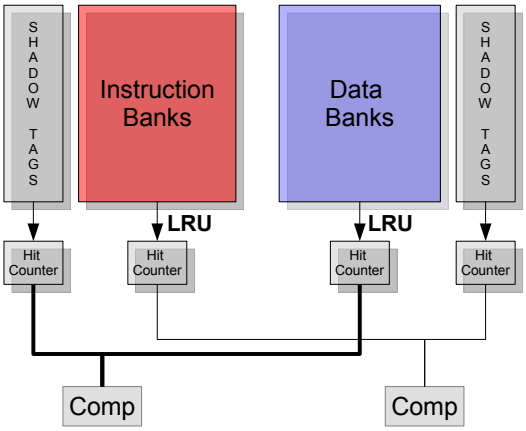


Figure 4.4: Shadow Tag VSC.

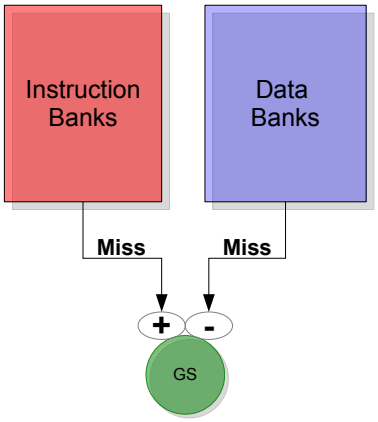


Figure 4.5: Global Selector VSC.

is then initialized with a value in the middle of its range. We have determined experimentally that a good range for the Global Selector counter is between 0 and twice the cache associativity minus one. This way this is the range used in our experiments, the reset value being the associativity. Figure 4.5 shows the structure of this design.

4.4. Simulation environment

To evaluate our approach we have used the SESC simulator [58] with a baseline configuration based on a four-issue out-of-order core with an hybrid branch predictor scheme as well as two on-chip cache levels. Both data cache levels use 32 MSHRs [41] while the instruction L1 uses 4. This configuration is detailed in Table 4.1.

We have used CACTI [21] to derive the latency related to each component of the memory hierarchy.

We have performed experiments using 13 benchmarks of the SPEC CPU 2006 suite. They have been executed using the reference input set (*ref*), during 10 billion instructions after the initialization. The results achieved under these conditions are very stable and representative of real behaviors. Table 4.2 characterizes them providing the miss rate for both instruction and data L1 caches, the combined miss

rate regarding the number of accesses and misses both caches have altogether and the CPI obtained with the baseline.

4.5. Experimental evaluation

We have applied VSC in the first level of the cache memory hierarchy to evaluate its performance in terms of IPC improvement and miss rate reduction. We have compared both versions of the VSC as well as a dual-ported unified cache with aggregated capacity, 64KB and 8 ways, with the baseline. The hit latency, calculated with CACTI [21], for this unified approach is 3 cycles. Both versions of the VSC have been evaluated using a variable latency depending on the number of banks allocated. Several research papers [2][6][87] as well as actual well-known architectures [75] have different latencies in the L1 cache despite the issue logic restrictions. Namely, we have estimated a latency for each one of the two virtual caches provided by the VSC of 1 cycle when it has 1 or 2 allocated banks, 2 cycles from 3 to 5 banks and, finally, 3 cycles if 6 or 7 banks are allocated. This latter latency of 3 cycles is actually an overestimation used to evaluate the design using whole numbers depending on the number of allocated banks, but 2 cycles would suffice in the worst case according to CACTI. For this reason we have also made evaluations of the VSC using a fixed hit latency of 2 cycles, which is the same used in the baseline configuration. The results of these experiments were very similar to those of the VSC with variable hit latencies, therefore they are not shown. Figure 4.6 shows the percentage of IPC improvement for the VSC designs and the unified cache related to the split baseline configuration. The unified cache degrades the IPC improvement related to the baseline in 1.5% due to its higher latency. Our Global Selector VSC gets a 3.2% improvement while using shadow tags this percentage is increased, up to 3.7%.

Figure 4.7 shows the percentage of miss rate reduction related to the combined miss rate of the baseline. The unified approach gets a 6% of miss rate reduction, the Global Selector VSC 11% and, finally, the Shadow Tag VSC 13%.

We can conclude that our VSC combines the lower latency of split approaches while providing even smaller miss rates than unified caches.

Table 4.1: Architecture. In the Table RT and TC stand for round trip and tag directory check, respectively

Processor	
Frequency	4GHz
Fetch/Issue	4/4
ROB entries	176
Integer/FP registers	96/80
Memory subsystem	
L1 i-cache & d-cache	32kB/4-ways/64B/LRU/WB
L1 Cache latency (cycles)	2 RT
L2 (unified, inclusive) cache	2MB/8-way/64B/LRU/WB
L2 Cache latency (cycles)	14 RT, 6 TC
Main memory latency	62ns

Table 4.2: Benchmarks characterization.

Benchmark	DL1 miss rate	IL1 miss rate	Combined miss rate	CPI
401.bzip2	2.8%	0.001%	1.8%	1.49
429.mcf	13%	0.001	8.9%	9.1
433.milc	5.7%	0.001%	3.7%	3.57
444.namd	1.7%	0.01%	1%	0.96
445.gobmk	1%	3%	1.7%	1.51
450.soplex	5%	1%	2.7%	2.32
456.hmmer	1.2	0.003%	1%	2.08
458.sjeng	1%	1%	1%	1.6
462.libquantum	7.8%	0.0001%	3.1%	2.85
470.lbm	20%	0.0001%	9.6%	2.08
471.omnetpp	6.8%	1%	6%	2
473.astar	5.2%	0.001%	3.1%	3.3
482.sphinx3	5.3%	0.01%	3%	2.94

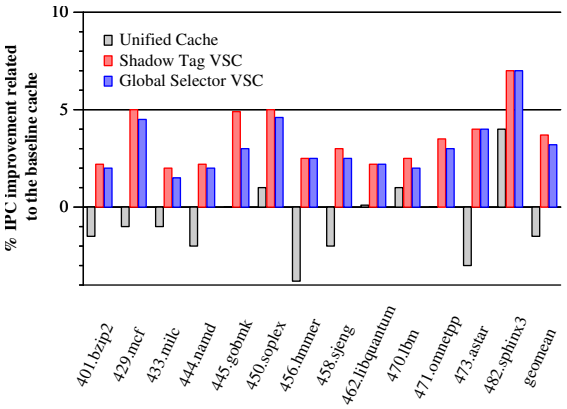


Figure 4.6: Percentage of IPC improvement related to the baseline cache.

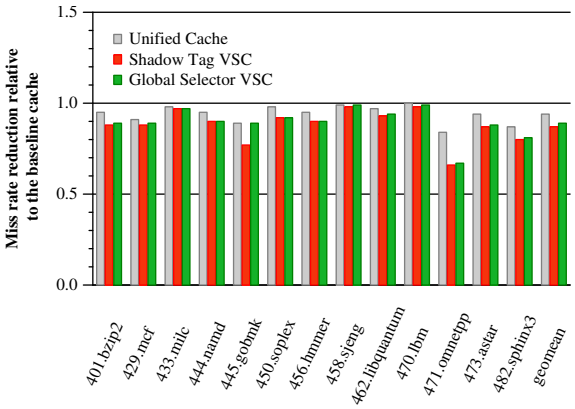


Figure 4.7: Percentage of miss rate reduction related to the baseline cache.

4.5.1. Average memory latency and power consumption

We have analyzed the average memory latency and the power consumption of our approaches related to the baseline configuration. The power consumption per access in the different levels of the hierarchy has been estimated using CACTI [21], the energy consumption in the first level of our VSC being slightly higher than that in the first level of the split baseline configuration. Figure 4.8 shows the relative average memory latency of the memory hierarchy when using the Shadow Tag and Global Selector VSC related to the one measured when the baseline split cache is used, and Figure 4.9 shows a similar study in terms of power consumption. Each bar is broken down in the percentage of hits that are satisfied in the first or second level of the memory hierarchy or in the main memory. The last column shows the arithmetic mean. As for the average memory latency, Shadow Tag VSC outperforms the baseline cache by 5% while Global Selector VSC does it by 4%. Moreover, despite the larger power cost of the VSC itself, when we look at the total power consumption of the memory hierarchy, the Shadow Tag VSC achieves 10% power consumption reduction with respect to the baseline design, while the Global Selector VSC provides an average 8% reduction. The reason for the larger reduction achieved by the Shadow Tag VSC is the larger fraction of accesses to the second level cache that it is able to avoid.

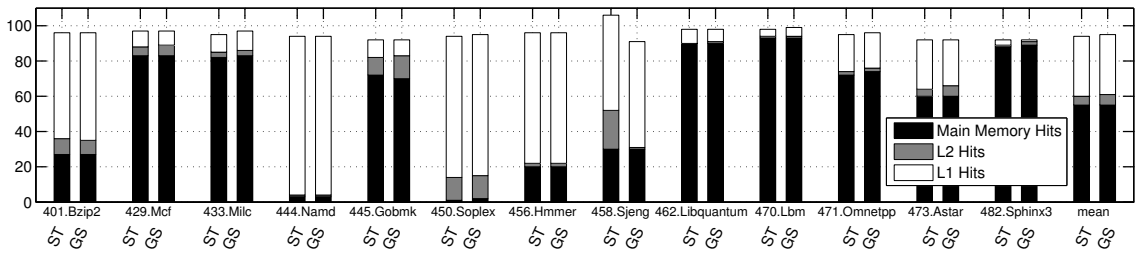


Figure 4.8: Average memory latency reduction of the Shadow Tag and Global Selector VSC relative to the split baseline.

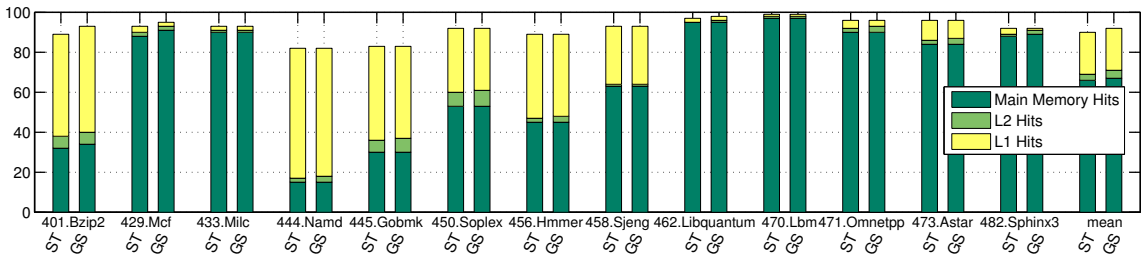


Figure 4.9: Power consumption of the Shadow Tag and Global Selector VSC relative to the split baseline.

Table 4.3: Baseline (Instructions cache + Data cache), Shadow Tag and Global Selector VSC storage cost in a 32KB/4-way/64B/LRU cache.

	Baseline	Shadow Tag VSC	Global Selector VSC
Tag-store entry:			
State(v+dirty+LRU)	1+1+2 bits	1+1+3 bits	1+1+3 bits
Tag ($42 - \log_2 sets - \log_2 64$)	29 bits	29 bits	29 bits
Size of tag-store entry	33 bits	34 bits	34 bits
Data-store entry:			
Set size	64*4*8 bits	64*8*8 bits	64*8*8 bits
Additional structs per set:			
Shadow Tags	-	2*29	-
Total of structs per set	-	58 bits	-
Total Counters:	-	16*3 + 19 bits	4 bits
N° of tag-store entries	512	1024	1024
N° of data-store entries	512	1024	1024
N° of sets	128	128	128
Size of the tag-store	2.0625KB	4.25KB	4.25KB
Size of the data-store	32KB	64KB	64KB
Size of additional storage	-	936B	4 bits
Total	(I+D) **2*34.0625KB**	∼**69.1KB** (∼1.4%)	∼**68.25KB** (∼0.1%)

4.6. Cost

In this section we evaluate the cost of both Shadow-Tag and Global Selector Virtually Split Caches in terms of storage requirements. Global Selector VSC only needs 4 additional bits, assuming an aggregated associativity of 8, for the saturation counter. Shadow Tag VSC needs two additional tags per set as well as two counters for tracking the hits in the shadow tags, either for instructions or data, and two additional counters in order to track the number of hits in LRU positions. According to our experiments and using the periodicity described in Section 4.3.1, the counter devoted to the number of hits in the LRU position for data needs 19 bits, while 16 bits are enough for the rest of the counters. Based on this, Table 6.4 calculates

the storage required by both approaches of the VSC and for a 4-way 32KB baseline cache with lines of 64B assuming addresses of 42 bits. Note that we have taken into account the storage cost for both instruction and data cache in the split baseline configuration. We can see that the storage cost of both approaches is small, even negligible in the case of the Global Selector VSC. We have also calculated that the storage cost of the Shadow Tag VSC could be halved, about to 0.65%, if we reduced the number of bits per tag to 10 bits applying the hash functions proposed in [57], since the number of lines that a set can hold, and consequently the number of possible values that a shadow tag can have, is limited.

4.7. Analysis

In this Section we analyze the internal behavior of our approach as well as the performance it achieves when the cache resources are shared among several cores.

4.7.1. VSC behavior

Figure 4.10 shows a box plot with the distribution of the number of banks allocated by the Shadow Tag VSC for both instructions and data during the execution of the benchmarks considered in Figure 4.1. The number of banks devoted to both instructions and data over time has been sampled each 1000000 accesses to the cache, i.e, each time a new adjustment is performed in the Shadow Tag VSC. The height of each box indicates the percentage of samples that lay between the first and third quartiles, while its width indicates the size of the sampled population. The median of the considered population is shown as a cross line in each box. Whiskers show both the minimum and maximum samples within the most significative population and, finally, circles, which colour gradually varies depending on the number of samples they embrace, represent outliers. We can observe how the poorer locality data have compared to instructions translates into all benchmarks allocating, in a certain moment of the execution, the maximum number of banks for data allowed by our approach, the associativity minus one. On the other hand, half the benchmarks allocate the maximum number of allowed banks for instructions in a certain moment of time, but most of them are outliers, even allocating from 4 or more ways

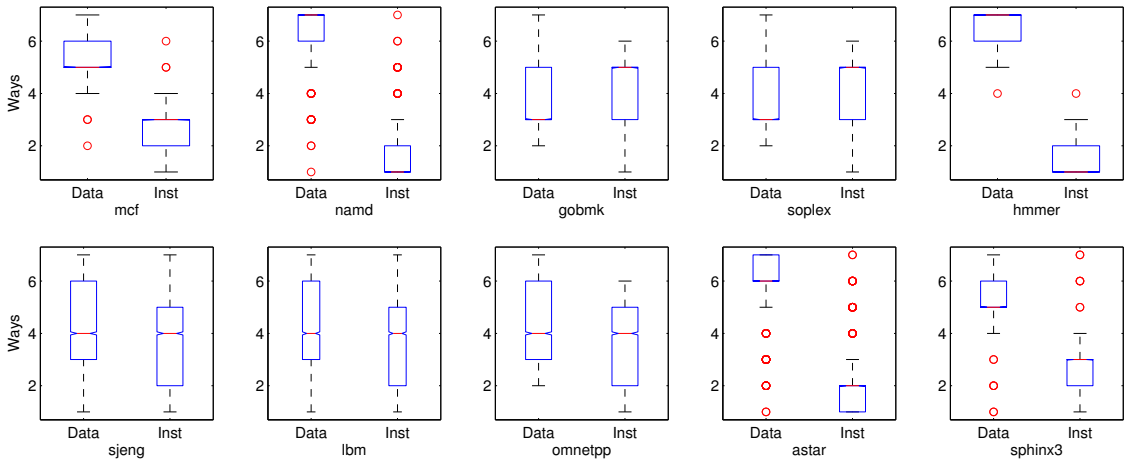


Figure 4.10: Distribution of the number of banks allocated for instructions and data using Shadow Tag VSC.

for instructions. As a result, all benchmark executions have always at least 3 banks allocated for data, existing some outliers with 1 and 2 allocated banks, while the minimum number of allocated banks for instructions is only 1, being 2 or 3 the common case. Only those benchmarks where the data working set is smaller than the instructions one, like *445.gobmk* or *462.lbm*, or where resources are fairly balanced between data and instructions, like *450.soplex* or *458.sjeng*, do not need to devote more banks for data than for instructions.

4.7.2. Multicore experiments

We have performed experiments in a multicore environment configured with a shared L2 cache of 4MB and 16 ways. We have used the benchmarks listed in Table 4.2 to test 16 multiprogrammed workloads of two applications and 6 of four. Each benchmarks is executed until it commits 10 billion instructions after the initialization, with the same reference input set as in the previous experiments. When each core reaches this number of instructions it continues its execution until the slowest core finishes, in order to keep competing for the cache resources. Figure 4.11 shows the percentage of IPC improvement related to the baseline for both Shadow Tag and Global Selector VSC. The last column indicates the geometric mean. Shadow Tag VSC gets a 4% IPC improvement while Global Selector VSC a lower 3.2%. As

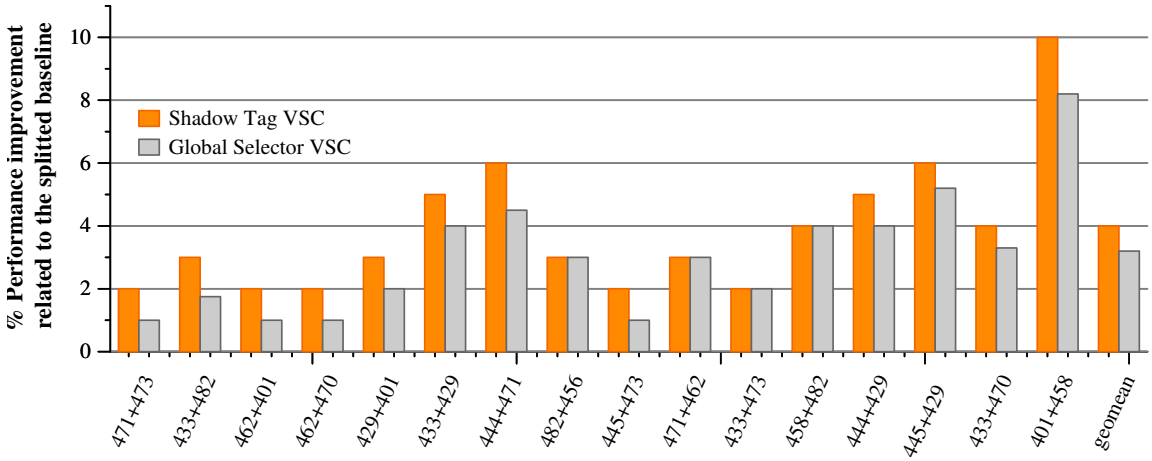


Figure 4.11: Performance improvement over the split baseline executing 2 applications using Shadow Tag and Global Selector VSC.

for the experiments executing 4 applications, shown in Figure 4.12, Shadow Tag and Global Selector VSC obtain 4.5% and 3.7%, respectively. From these results we can infer that the benefits of our VSC increase as the number of cores applying it and sharing the lower level increases. As our VSC has been successfully proved to reduce the miss rate, the number of accesses that the shared lower level must handle

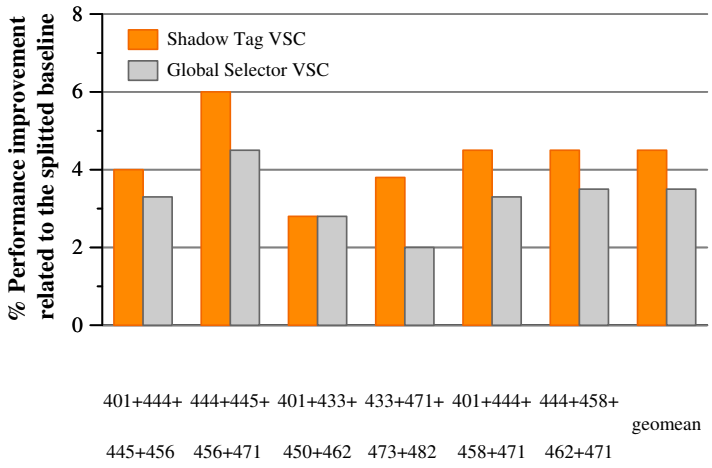


Figure 4.12: Performance improvement over the split baseline executing 4 applications using Shadow Tag and Global Selector VSC.

is lower, which increases performance.

4.8. Summary

This chapter introduces the *Virtually Split Cache* (VSC), a new design specifically oriented for the first level of the cache memory hierarchy that provides higher hit rates than split configurations and lower latency than unified approaches at the same time by balancing resources between instructions and data. It is aware of the different locality instructions and data have and it allocates resources for both depending on their demand. Two alternative designs to track the different requirements instructions and data demand are proposed. The first approach, the Shadow Tag VSC, uses shadow tags to decide whether assigning one more bank for instructions or data increases performance. The second approach, the Global Selector VSC, uses a common saturation counter to make instructions and data fight a duel for resources.

Shadow Tag VSC achieved 3.7% IPC improvement and 13% and 10% of miss rate and power consumption reduction related to a split baseline, respectively. Global Selector VSC got 3.2% IPC improvement and 11% miss rate reduction needing only 4 bits of additional storage, while achieving 8% of power consumption reduction as well. Furthermore, both approaches achieve notable reductions in power consumption and have been successfully proved to work well in multicore environments. Shadow Tag VSC outperformed a 4-core baseline configuration by 4.5% on average in terms of throughput while the Global Selector VSC did it by 3.7%.

Chapter 5

Thread-Aware Bimodal Set Balancing Cache

5.1. Introduction

In an attempt to make a more efficient usage of their caches, the memory hierarchies of many chip multiprocessors (CMPs) present last-level caches which are shared by several cores, thus facilitating the dynamic allocation of their resources among the cores. Unfortunately, the standard management strategies for private caches, which are by nature thread-oblivious, often lead to suboptimal behaviors when applied to shared caches. This fact has been largely recognized and studied, and a wide variety of techniques to improve the performance, fairness and Quality of Service (QoS) of shared caches have been proposed. Many approaches focus on the distribution of the cache resources among the competing applications [13][54][74], while other ones explore the adaption of cache policies to the behavior of each application [27], there being also mixed proposals like [84]. Finally, there are also techniques, which being oblivious to the behavior of each application, and thus also applicable to non-shared caches, have proved to benefit shared caches [9][36][37][82]. In Chapter 3 we introduced the Bimodal Set Balancing Cache (BSBC), an approach specifically designed to reduce capacity and conflict misses at the same time in single core environments, thanks to the Set Saturation Level (SSL) metric versatility. This technique proved to be very beneficial in shared caches as well, although thread-aware techniques

showed that there is still room for improvement in this kind of environment.

In this chapter we extend the scope of application of the Set Saturation Level (SSL) metric by proposing and evaluating a new strategy for shared caches which takes some decisions based on the joint behavior of all the applications that share the cache, while other decisions are specific to the particular behavior of each application. Still, these latter decisions do not consider the application in isolation, but rather take also into account how the other applications impact on the behavior of the analyzed one. This new proposal considers both the potential unbalance in the distribution of the memory references and the existence of working sets which cannot be accommodated in the cache, and gives an appropriate answer to them in a coordinated way. It is worthy to point out that these problems are exacerbated in shared caches, where the capacity available to each application is diminished, and where thrashing applications hurt not only themselves, but the other sharers as well.

Our technique for shared caches, called Thread-Aware Bimodal Set Balancing Cache (TABSBC) detects the degree to which a given thread can suffer these problems in each cache set independently and applies the best combination of policies to the lines of each thread at a set level by using the SSL metric.

5.2. Background and Motivation

There have been many proposals [8][17][25][54][70] specifically designed to optimize the behavior of shared caches by partitioning their resources among the applications that share them. A problem with partitioning is that, depending on its granularity and strictness, it can lead to resource underutilization. For example, in a way-partitioning scheme it can well happen that the lines reserved to a given core in some sets are of little or even no use, while other cores struggle to keep their working set in those sets.

If we consider the traditional 3C classification of the cache misses, two kinds of problems avoid the successful reuse of lines: the restrictions due to the placement algorithm (conflict misses), and the lack of space to hold the working set (capacity misses). In the scope of shared caches, we can further distinguish between self and inter conflict and capacity misses. Self misses would be the ones that take place even

if the associated application had all the cache for itself, while the inter-conflict and inter-capacity misses would be the new misses of the respective kind that appear in the presence of other applications, considering together all their working sets. Detecting the conflict and capacity misses of shared caches and addressing them specifically seems thus a good management strategy.

A technique that follows this approach is Adaptive Set Pinning (ASP) [70], which reduces inter-conflict misses according to our classification thanks to the ownership of each cache set by a processor, which is the only one that can insert new lines in it, the other processors having to resort to a small processor owned private (POP) cache partition. It cannot help with self-conflict misses, though.

A proposal oriented to self and inter-capacity misses is the Thread-Aware Dynamic Insertion Policy (TADIP) [27], which extends to shared caches the insertion policies introduced in [53] to deal with capacity misses in private caches. TADIP can apply these policies in isolation to each independent thread according to the benefit it can get from them, measured by means of set dueling. The thread-aware TA-DRRIP [28] chooses the most appropriate Re-Reference Interval Prediction (RRIP) insertion policy for each application based on set dueling, RRIP being mainly focused on capacity misses, as we explained in Section 3.2. The Pseudo-LIFO policies [9] are also particularly suited to capacity misses. These policies evict blocks from the upper part of the fill stack, that is, among the most recently inserted lines of the set, thereby retaining a large fraction of the working set in the cache. Dead-block prediction based replacement and bypass policies [37], which try to avoid polluting the cache with dead blocks by evicting them as soon as possible or even bypassing them help mainly with the reduction of capacity misses as well.

Furthermore, there are approaches like Promotion/Insertion Pseudo-Partitioning (PIPP) [84], which combines pseudo-partitioning with new insertion and promotion policies. Namely, if the partitioning algorithm it relies on, which is Utility-based Cache Partitioning or UCP [54], assigns n ways to a thread, its insertions take place n positions away from the bottom of the recency stack, the top being the MRU line. Also, hits do not promote lines to the MRU position, but a single position up in the stack, and only with a given probability. This, coupled with a variation for applications in streaming mode, gives PIPP special protection against capacity misses.

As we see, there are no proposals to our knowledge that tackle specifically both the self and inter-conflict misses as well as the self and inter-capacity misses we identify in shared caches. Also, most of the proposals that target capacity problems in shared caches take their insertion decisions globally, based on an average picture of the cache behavior obtained either from set dueling [53] or set sampling [54]. As a consequence, they apply those insertion decisions uniformly in all the cache sets.

5.3. Thread-Aware Bimodal Set Balancing Cache

When the techniques specifically designed for single core or private LLCs are applied in shared caches, they are not aware of the interaction between several threads in the cache. We will consider in turn the detection of the two problems to retain the working set identified in Section 5.2 and the strategies to deal with them in shared caches, followed by the coordinated implementation we propose. Specifically, our approach coordinates in a sensible way the previously described SBC, which mainly deals with conflict misses, with a modified BIP, which we have called BIP-C, to tackle capacity problems by using the SSL metric to control both behaviors.

5.3.1. Unbalances among sets: Conflict Misses

When several applications share a cache, each set should hold the working set of all of them, and unbalance among sets can be exploited only if there are sets that still have underutilized lines after considering together the working set of all the applications. Thus, in principle it would not be of practical interest to distinguish between self and inter-conflict misses. We have considered the Dynamic SBC, due to its flexibility and low complexity, in order to deal with unbalances among sets. A single SSL per set would be an useful indicator of the joint pressure on the set. Therefore, our proposal for shared caches relies on SSL to detect load unbalance among sets and applies modified SBC policies, which will be described in Section 5.3.4, to solve them.

5.3.2. Lack of space in the cache: Capacity Misses

An application experiences capacity misses when its working set cannot fit in the cache even if the restrictions of the placement algorithm are raised. Since we use SSL and SBC to deal with conflict misses, a situation of lack of capacity of the cache can be detected by noticing that the SBC displacements of lines cannot be applied or do not suffice to solve the stress in the cache sets. As we have seen in Chapter 3, a clear indicator of this situation is the impossibility of finding appropriate destination sets when a non associated set reaches the highest value in its saturation counter applying SBC. This means that all the other sets are either already involved in associations, or they have a SSL too high to become destination sets. When this happens, a policy suitable for capacity misses should be applied to the set. While in the BSBC design the insertion policy for sets with the highest SSL that could not find a destination set was changed to BIP, in our TABSBC design for shared caches we will use a modified version called BIP-C which will be described in Section 5.3.3.

There are two main problems left to be solved. The first one is how to return to MRU insertion after switching to BIP-C if the working set evolves to fit in the cache. If this happens, the SSL will decrease gradually thanks to the higher hit rate. Thus, a simple solution is to revert to MRU insertion when the SSL falls below K, as the BSBC does.

The second problem is that while the resolution of conflict misses displacing lines from overloaded sets to underutilized ones without distinguishing the owner of the displaced line seems sensible, this is not the case for the modification of the insertion policy of different applications to reduce capacity misses. Applications with reasonable amounts of locality will be hurt by BIP. The Thread-Aware Dynamic Insertion Policy (TADIP) [27] has already shown the importance of this topic. TADIP uses independent Set Dueling Monitors for each application sharing the cache in order to discover the policy that suits it best in the presence of the other applications, achieving substantially better results than applying BIP or MRU to all the threads disregarding their individual characteristics.

The Set Dueling approach of TADIP leads to take global decisions for the whole cache, that is, the insertion policy it chooses is applied to all the sets. This is sensible, since it deals mainly with capacity problems, which in general affect the

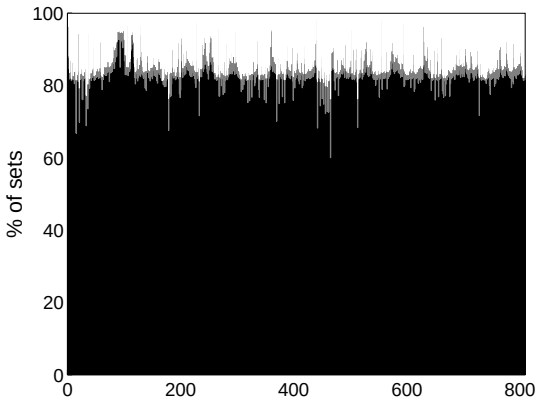


Figure 5.1: Distribution of the sets with a high (black), medium (gray) and low (white) SSL for SPEC CPU 2006 application *433.milc* during its simultaneous execution with SPEC CPU 2006 application *482.sphinx3* in a 8-way 2MB cache. Samples each $5 * 10^5$K accesses.

whole cache. But in our case some sets could solve their capacity problems thanks to displacements to underutilized sets. Also it is possible that while a majority of the sets are under pressure, a minority work well under MRU insertion, being counterproductive to switch them to BIP. Figure 5.1 illustrates a situation we have seen in many parallel executions. It classifies the sets of a 8-way 2MB L2 cache shared by SPEC CPU 2006 applications 433.milc and 482.sphinx3, during the first $400 * 10^6$ accesses to the cache in three categories. The sets are classified according to the value of an SSL which is only updated by the accesses by benchmark 433.milc to the set. Concretely, an SSL is classified as low (0-5), medium (6-10) or high (11-15). Another SSL updated in each set only by 482.sphinx3, not shown, is uniformly saturated in all the sets along this simultaneous execution. Nevertheless we see there is a representative and sustained 20% to 25% of sets in which 433.milc exhibits a low SSL, pointing to a good locality that should benefit from MRU insertion. As a result of the increasing number of cores in current systems, it is of the upmost priority to reduce as much as possible the bandwidth requirements of shared LLCs, provided the hardware complexity involved is within reasonable margins. Thus, given these observations, we use one SSL per core per set to choose the appropriate insertion policy with the best granularity.

5.3.3. BIP-C

Our proposal uses a modified BIP, which we call BIP-C, that inserts the incoming lines not in the LRU position of the recency stack, but $min\{C-1, \lfloor K/2 \rfloor\}$ positions away, where C is the number of cores sharing the cache and K the associativity. The rationale for this is that while in a private cache it is only up to the owner application to evict the line with a subsequent miss, or reuse it and bring it to the MRU position, in a shared cache any other application can evict the line before the owner has a chance to reuse it. The situation is even more challenging when the cache works under our TABSBC proposal, as lines near the bottom of the LRU stack can be replaced not only by incoming lines mapped to the set, but from lines that come from displacements from source sets in associations as a result of applying SBC. Our approach inserts deeply lines in the recency stack in typical configurations (K $\gg$ C) while allowing a few misses in the set from the other running threads before the line is evicted, so that the owner has more chances to prove the merit of the line. Figure 5.2 shows an example of its operation in a LLC shared by 2 cores and assuming a replacement operation driven by core number 1. In the following we will say an application is in BIP-C mode in a set when its insertions apply this policy, and in MRU mode otherwise. Figure 5.3 shows the throughput improvement of TABSBC with respect to a baseline 4MB 16-way cache shared by four cores when using five BIP variations in the presence of capacity problems. The variations are the original BIP, BIP-CC, which inserts new lines as many positions away from the LRU one as different cores have lines in the set; BIP-C/2, which inserts $min\{\lfloor C/2 \rfloor, \lfloor K/2 \rfloor\}$ positions away from the LRU one; BIP-2*C, which inserts new lines upper in the stack, specifically $min\{2*C, \lfloor K/2 \rfloor\}$ positions away from the LRU; and finally, BIP-

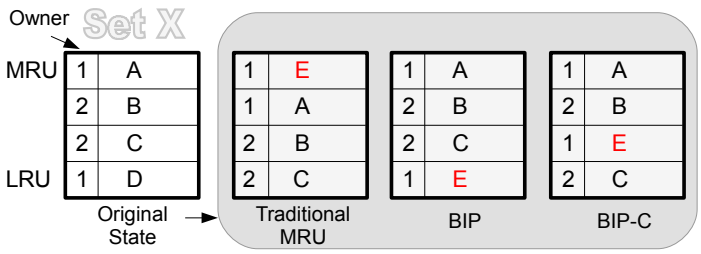


Figure 5.2: Behavior of different insertion policies after inserting new line *E* in set *X* of a 4-way LLC shared by *2* cores.

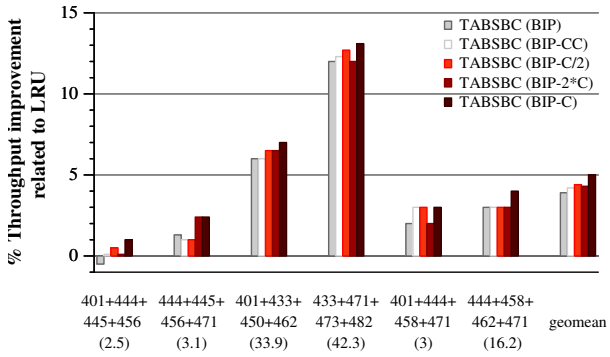


Figure 5.3: Throughput improvement over a 4MB 16 ways baseline cache shared by 4 cores using TABSBC with different versions of BIP. MPKI of each workload in parenthesis under its name.

C. The workloads consist of SPEC CPU 2006 benchmarks that will be presented in Section 6.5 with all the other parameters of the simulations. BIP-C outperforms clearly the other approaches. Altogether, TABSBC using BIP-C provides 1.1% better throughput than a modified TABSBC using BIP due to the reasons explained above.

5.3.4. Computing Set Saturation Levels

Section 5.3.1 argued that a single SSL counter per set is enough to detect unbalance between sets, while Section 5.3.2 argued on the need to have one SSL per set per core in order to learn the best insertion policy for each application. Thus TABSBC uses one SSL per set per core. Each time an access updates a SSL, a Global SSL (GSSL) is computed for the set. This value represents the global capacity of the set to hold the particular working sets of all applications sharing the cache. The GSSL does not need to be stored in the set; it is just supplied to the DSS (see Section 2.4.1). Thus, if the GSSL computed is smaller than the maximum value in the selector, and it is also below the saturation limit K (the cache associativity) allowed for a set to be candidate to become a destination set, the set index and the GSSL are stored in the DSS.

The GSSL can be computed by simply adding the SSLs and saturating the outcome to the maximum value of a single SSL. The GSSL can be also tuned taking into account the insertion policy of each application. Applications in BIP-C mode

always have a SSL$\geq K$, since they return to MRU mode when the SSL falls below K, as this indicates that the working set fits in the set. This way, if their SSLs are added up directly, these applications preclude by themselves GSSLs smaller than K, making impossible for the corresponding sets to become destination sets. Still, when an application is in BIP-C mode, its high SSL does not mean that it is using effectively many lines in the set. Rather it means that there is high recent miss rate, probably because the application is memory-intensive. As as result it is quite probable that many of the lines that belong to the application in the set are of little use and it would be better to give them to another application. Of course this can also happen with applications with MRU insertion, but the chances are higher among those with BIP-C. Thus in our experiments the SSL of applications in BIP-C mode is scaled down for the sake of the computation of the GSSL by dividing them by 2, since this can be implemented in hardware with a simple shift of one bit. This minor change improved the throughput in our tests around 0.25%.

5.3.5. Interaction between the Insertion and the Placement Policy

1. ***When to switch the insertion policy of a set to BIP-C?*** If a certain application has capacity problems despite applying the previously described policies, the associations established initially will become useless. In fact they will be counterproductive because dead lines will be moved to destination sets, giving later place to second searches that will be useless most of the time, thereby delaying the resolution of misses and potentially increasing the miss rates in the destination sets.

2. ***When to switch the insertion policy of sets involved in associations?*** Also, there must be a way to enable an application to change to BIP-C in the sets that participate in an association, and eventually break it, if the high SSLs continue. TABSBC does this by changing the insertion policy to BIP-C also for (1) applications in destination sets in which their SSL reach the highest value, and (2) applications in source sets that try to displace a line to their destination set and find that the application follows BIP-C there. Notice that the second situation implies that the SSL of the application has also the

highest value in the source set. This way, if an application suffers a capacity problem in an association, this is first detected in the destination set, which is the one that suffers more pressure because of the displacements, and then it is propagated to the source set. Relatedly, in TABSBC if an application follows BIP-C in a source set, it stops displacing lines to the destination set, since the displacements are not helping. This avoids the counterproductive situation described above. It also avoids the perverse effect that lines recently inserted in the bottom of the recency stack of the destination set by BIP-C are evicted due to displacements from the source set before having a chance to be reused. Further, this strategy facilitates breaking the association, since this happens when the destination set evicts all the lines from the source set. Misses in the source set always lead to searches in the destination set while the association lasts.

3. ***How are the displacements of lines between sets triggered?*** While it is clear that insertion is ruled by the application that requests the line to be inserted, a clarification is needed on the rules for displacements. TABSBC, as the SBC, displaces the LRU line of the source set, as it is the line selected by the replacement policy. Nevertheless its displacement must not be ruled by the application that generates the eviction, but by the owner of the line to be potentially displaced. Thus TABSBC requires storing the owner of each line ($\log_2$C bits, where C is the number of cores) in the tag-store. Under a miss, the field is examined for the LRU line in a set to decide which is the policy to apply based on the insertion mode and SSL of its owner.

Finally, the association algorithm changes slightly with respect to the one in SBC. Association attempts are triggered when under a miss the owner of the LRU line is in MRU mode and its SSL is the highest one. This is sensible since this is a precondition for a displacement to take place. The association takes place if the DSS can provide a suitable destination set, that is, one with a GSSL smaller than K.

Figure 5.4 shows a C-like pseudocode of the actions taken by TABSBC under an access to a set. In the figure, SSL represents a vector with one value per core, which means the SSL for each core in the accessed set. The computation of the GSSL and

```
access(Core i, Address addr) {
 if hit(addr) {
   SSL[i]--;
   move addr line to MRU;
   if lowlySaturated(SSL[i])
     mode[i] = MRU;
 }
 else {
   SSL[i]++;
   if (this is a secondary search)
     return;
   if (set is source set of an association)
     if (access(i, addr) is successful in destinationSet)
       return;
   request addr line to memory;
   let lineLRU = line in LRU position in set;

   if (set is not associated) {
     if (exists candidate destination set in DSS) {
       if mode[lineLRU.owner] == MRU && saturated(SSL[lineLRU.owner])
         associate this set to candidate destination set;
     } else if saturated(SSL[i])
         mode[i] = BIP-C;
   }

   if (set is source set of an association
       && mode[lineLRU.owner] == MRU
       && saturated(SSL[lineLRU.owner])
     if destinationSet.mode[lineLRU.owner] == BIP-C {
       mode[lineLRU.owner] = BIP-C;
       evict lineLRU from cache;
     } else
       displace lineLRU to MRU position in destination set;
   else
     evict lineLRU from cache;

   if (set is the destination set of an association) {
     if saturated(SSL[i])
       mode[i] = BIP-C;
     if (there are no lines from the source set)
       break association;
   }
 }
}
```

Figure 5.4: TABSBC operation under a cache access.

corresponding update of the DSS, which take place when an SSL is updated, are elided. The insertion of incoming lines, which simply follows the insertion policy of the owner application in the set, is not reflected either because it takes place when they arrive.

5.3.6. Contribution of each policy to TABSBC performance

TABSBC coordinates techniques to improve the cache behavior in a thread-aware way. In order to justify our design decisions, Figure 5.5 analyzes the contribution of each portion of TABSBC to its global behavior by comparing the average throughput improvement that several policies achieve over the 4MB 16-ways baseline cache shared by two cores described in Section 5.4 when running the workloads in Table 5.1. From left to right, a non-thread aware Dynamic Insertion Policy (DIP) [53], which sets the whole cache in BIP or LRU insertion mode, provides less performance than a non-thread aware SBC. The Bimodal Set Balancing Cache (BSBC), which is unaware of the existence and the behavior of the different applications, performs much better by coordinating efficiently placement and insertion policies to reduce conflict and capacity misses. TADIP thread-awareness applied to insertion policy management brings large advantages in shared caches, as seen in [27].

The next policy evaluated, TASBC, is a thread-aware SBC, that is, it is similar to TABSBC but without applying any insertion policy to tackle capacity problems. TASBC has no advantages over SBC because the problem solved by SBC, i.e., the unbalance of the load of different cache sets, is not specific to the behavior of a thread, but to the combined behavior of all of them in each set, as Section 5.3.1 explains. The importance of coordinating adequately the insertion and placement policies can be seen by comparing BSBC, which is totally thread-oblivious but coordinates the insertion policy and SBC, with DIP+TASBC, which applies independently DIP and a thread-aware SBC, being each one of them totally unaware of the behavior of the other one.

Applying TADIP with TASBC in an uncoordinated way brings similar performance advantages to those of discovering and applying the best insertion policy to each thread in each set independently using the SSL metric instead of *set dueling*, a technique we have called TADIP-Local. If TADIP-Local is combined with SBC

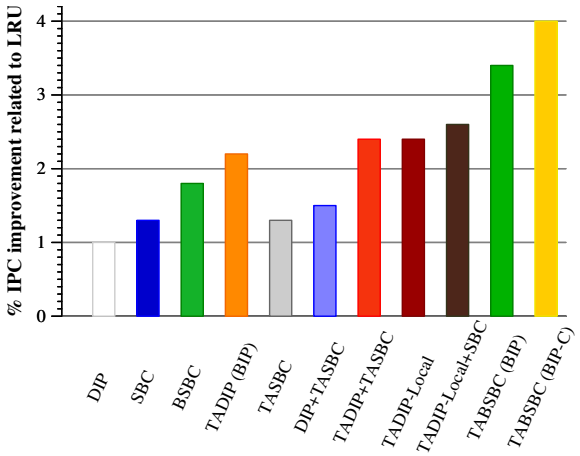


Figure 5.5: Throughput improvement over a 4MB 16 ways baseline cache shared by two cores under several policies.

(which, let us remember, has very similar performance to TASBC) in an uncoordinated way, a small additional performance gain is achieved. Let us recall that all those approaches that combine in a non-coordinated way a technique that deals with capacity problems, like DIP, with another one that mainly tackles conflict misses, like SBC, can generate harmful behaviors like displacing recently inserted lines in source sets working under BIP, giving raise to unsuccessful second searches, or like evicting just inserted lines in destination sets due to displacements, depriving them of a chance to be reused before being evicted. They have an aggregate storage overhead as well. This way, better results are obtained when TADIP-Local+SBC is improved implementing the coordination provided by the TABSBC, giving place to TABSBC (BIP), which uses BIP for the applications that suffer of capacity problems, just like TADIP and TADIP-Local. Finally, when TABSBC applies the BIP-C policy proposed in this paper, we reach the maximum performance.

The relative contributions of each policy vary of course with the environment. For example thread-awareness relevance grows with the number of cores sharing the cache, while SBC policies become more important for workloads whose working sets fit relatively well in the cache, for which MRU insertion works well.

5.4. Simulation environment

We use the SESC simulator [58] and the two-level baseline configuration described in Chapter 2 with some adjustments to evaluate our proposal. The baseline system consists of two cores with private L1 caches and a shared L2 cache. The size of this cache and its associativity have been increased up to 4MB and 16 ways, respectively. We have used the same workloads as in Section 3.7.3 and under the same conditions. The statistics for each application are only recorded during the execution of its first 10 billion instructions. These long simulations, in which applications can go through several stages, lead to lower IPC improvements over the baseline for all the policies in our experiments than those observed in other studies. We feel the results achieved in this way are very stable and representative of real behaviors. Table 5.1 characterizes the multiprogrammed workloads with the miss rate and MPKI of each combination in the 4MB 16-way L2 shared cache.

5.4.1. Metrics

Our evaluation relies on metrics usually referenced in the bibliography: Throughput, Weighted Speedup [67], which indicates execution time reductions, and the Harmonic mean of weighted speedups [48], which balances fairness and performance. Concretely, being N the number of applications of the multiprogrammed workload and assuming one application per core, IPC_i the IPC achieved by application i when running concurrently with the others in the workload, and $SingleIPC_i$ the IPC of application i when running in isolation:

$$\textbf{Throughput} = \sum_{i=1}^{N} IPC_i$$
$$\textbf{Weighted Speedup} = \sum_{i=1}^{N} (IPC_i/SingleIPC_i)$$
$$\textbf{Harmonic mean of normalized speedups} = N/\sum_{i=1}^{N} (SingleIPC_i/IPC_i)$$

Table 5.1: Multiprogrammed workloads characterization.

Name	Benchmarks	L2 Miss rate	MPKI
MW1	471.omnetpp + 473.astar	7.8%	6
MW2	433.milc + 482.sphinx3	65.1%	26
MW3	401.bzip2 + 462.libquantum	32.3%	13.2
MW4	462.libquantum + 470.lbm	36.1%	45.6
MW5	401.bzip2 + 429.mcf	17.5%	24
MW6	429.mcf + 433.milc	49%	72
MW7	444.namd + 471.omnetpp	1.6%	1.1
MW8	456.hmmer + 482.sphinx3	52.4%	10.4
MW9	445.gobmk + 473.astar	12.9%	4.7
MW10	462.libquantum + 471.omnetpp	24.5%	13.4
MW11	433.milc + 473.astar	42.1%	19.2
MW12	458.sjeng + 482.sphinx3	52.4%	9
MW13	429.mcf + 444.namd	15.4%	18.2
MW14	429.mcf + 445.gobmk	18.3%	21
MW15	433.milc + 470.lbm	37.5%	56
MW16	401.bzip2 + 458.sjeng	6.5%	1.2

5.5. Experimental evaluation

TABSBC, as well as the other approaches, has been applied in the shared last level of the cache memory hierarchy. As for the parameters that are specific to the different approaches evaluated in this study, TADIP (specifically TADIP-Feedback) [27] and TADRRIP (using *Hit Priority*) [28] use 32 sets dedicated to each policy for each core to decide between BIP and MRU insertion in the former case, and between BRRIP and SRRIP in the latter. These bimodal policies, BIP and BRRIP, as well as the one triggered by TABSBC, use a probability $\varepsilon = 1/32$ that a new line is inserted in the MRU position of the recency stack or with a long re-reference interval prediction in the BRRIP case. PIPP [84] uses 32 dedicated sets to track the hit counters information in the utility monitors and a probability of promotion of 3/4 and 1/128 in normal and streaming mode, respectively. The cache switches to streaming mode if the number of misses per interval is greater than 4095 or the miss rate in the shadow tags exceeds $12,5\%$. The probabilistic escape LIFO we evaluate, or pseudoLIFO in what follows, approximates the hit counts by the next power of two for escape probabilities and uses 4 dedicated sets every 1024 sets in the cache for each one of the four policies described in [9]. Finally, we also include the BSBC. TABSBC and BSBC use a Destination Set Selector of four entries like

the one used in the previous chapters.

Figure 5.6 shows the throughput improvement of all the approaches related to the LRU traditional policy for the 16 workloads of 2 applications described above. In this figure and all the subsequent ones the last group of columns corresponds to the geometric mean for the values achieved for all the workloads. This mean improvement achieved by each technique is: BSBC 1.8%, PIPP 2%, PseudoLIFO 1.2%, TADIP 2.2%, TADRRIP 3.1% and 4% for the TABSBC. These results show two trends that appear consistently in all the evaluation. The first one is that, non surprisingly, thread-aware approaches, tend to outperform in general the non-thread-aware BSBC and PseudoLIFO. The advantage of thread-awareness can be also observed comparing BSBC and TABSBC: out of the 16 workloads, the thread-aware TABSBC technique proposed in this paper outperforms the thread-oblivious BSBC, which also reduces capacity as well as conflict misses, in 11 workloads, while both yield very similar results in the other 5 workloads. Further, TABSBC improvements over the baseline are on average 120% larger than those of BSBC. The second trend observed is that those approaches that tackle both conflict and capacity issues tend to achieve the best results in their category. The other techniques can reduce capacity misses. Some also improve the detection and eviction of dead blocks in each independent set, which can reduce conflicts, but they cannot exploit the load unbalance among sets, therefore it is more difficult for them to make effective use of large portions of dead cache lines [36].

The studies for the other metrics related to speedup show very similar trends to those observed in Figure 5.6 for the throughput improvement. Therefore, Figure 5.7 (a) and 5.7 (b), devoted to the weighted speedup and the harmonic mean of weighted speedups, respectively, only show the values for BSBC and TABSBC in order to emphasize the value of thread-awareness. This way, under the weighted speedup metric the relative advantage of TABSBC over the baseline is about 100% larger than that of BSBC, since TABSBC is able to apply the best policies to each thread depending on its behavior. This helps avoid slowing down threads at the expense of others.

Our fairness metric also reflects clearly the positive properties of TABSBC in this regard. Figure 5.7 (b) shows the percentage of improvement of the harmonic mean of normalized speedups over the one measured for the baseline 16-way 4MB cache.

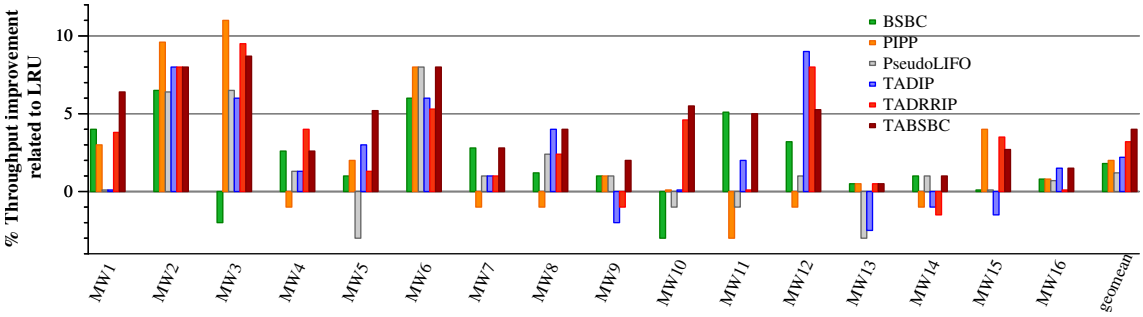


Figure 5.6: Percentage of throughput improvement over the 4MB 16 ways baseline configuration using several policies.

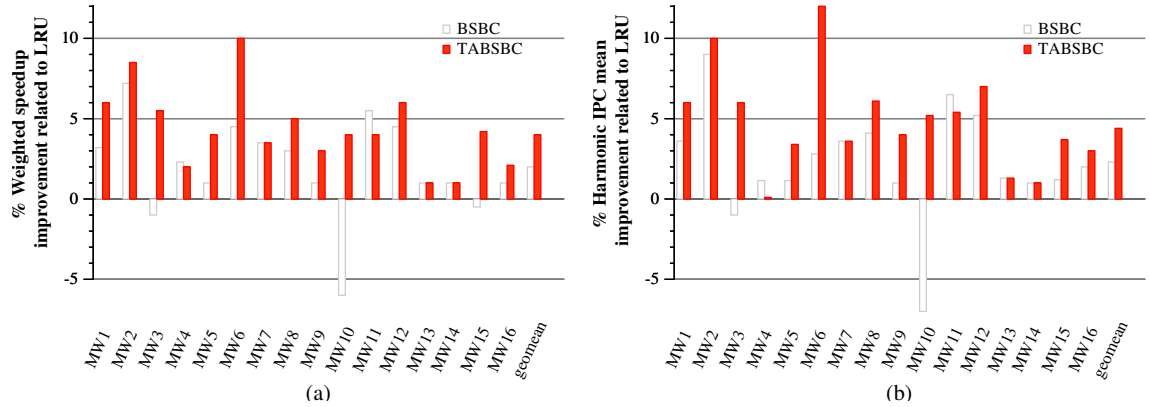


Figure 5.7: Percentage of weighted speedup (a) and harmonic IPC (b) improvement over the 4MB 16 ways baseline configuration using BSBC and the TABSBC.

On average, BSBC gets 2.3% improvement over the baseline and TABSBC 4.4%. The average values for the other techniques evaluated not shown in the graph are: PIPP 0.75%, PseudoLIFO 1.5%, TADIP 2.6%, and TADRRIP 3.4%. As expected, in general, thread-aware policies show more potential for fairness. Among them, TABSBC shows the overall best behavior because while the other techniques are restricted to managing the working set associated to each cache set, TABSBC can promote better a fair usage of the cache resources thanks to the incorporation of a displacement policy among sets based on their state.

It is interesting to notice that TABSBC is the only proposal that does not slow down any IPC metrics for absolutely any workload in any configuration with respect to its baseline. This is in contrast with other strategies, which can reduce the metrics up to 10% (15% in the case of fairness) in some experiments.

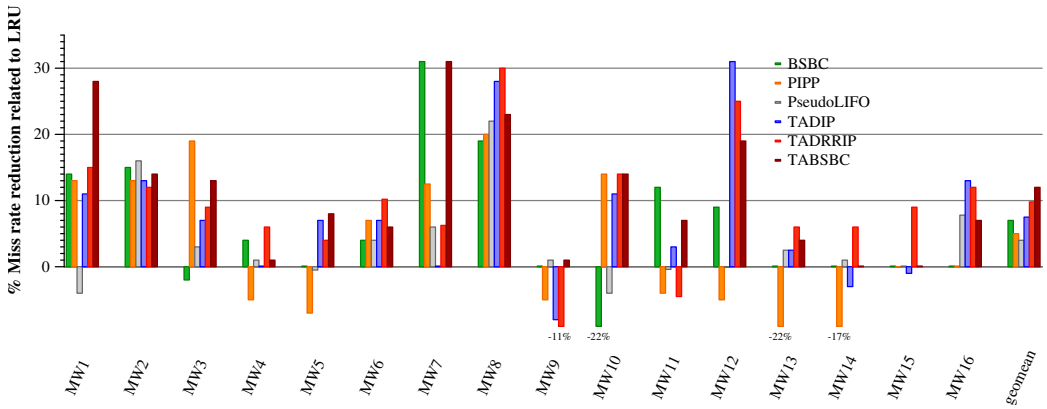


Figure 5.8: Miss rate reduction over the 4MB 16 ways baseline configuration using several policies.

Our last metric is the global miss rate, which considers together, as a whole, the accesses of all the applications during their simultaneous execution until the slowest one completes its 10 billion instructions. That is, it includes also the accesses of the fastest applications after they complete their 10 billion instructions. These rates give an idea of the reduction of bandwidth to memory provided by each approach, a resource which becomes more critical as the number of cores in CMPs increases. Since accesses to memory require much more power than cache hits, they are also a good indicator of the energy savings that can be achieved by the different techniques. Figure 5.8 shows this miss rate reduction related to the one observed in the baseline for the 4MB L2 caches considered. The (geometric) average relative reduction of the miss rate achieved by each policy is BSBC 7%, PIPP 5%, PseudoLIFO 4%, TADIP 7.5%, TADRRIP 9.8% and TABSBC 12%. The qualitative tendency in these figures is the same observed for the IPC metrics, the advantage of coordinated approaches to reduce conflict and capacity misses being larger. This way, BSBC can match and even outperform techniques specifically designed for shared caches. When the benefit of thread-aware policies is added, the good miss rate reduction achieved by SBC can still be almost duplicated by TABSBC, as we see in Figure 5.8. In fact, TABSBC, with only a 0.65% overhead as we will see in section 6.8, provides 40% of the miss rate reduction achieved by doubling the size of our baseline cache to 8MB.

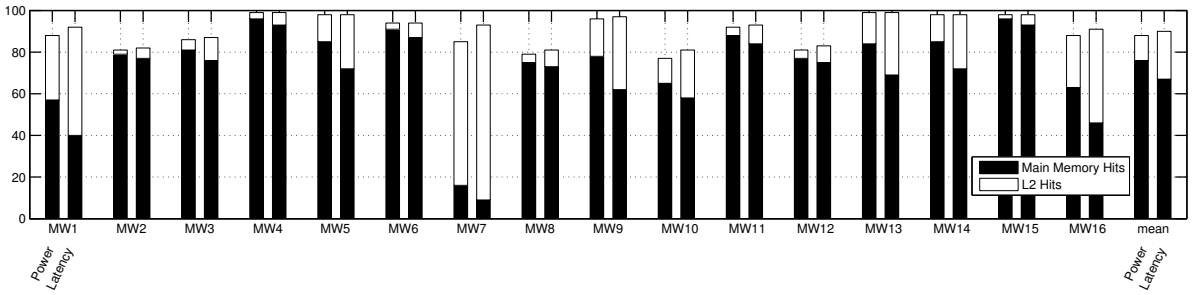


Figure 5.9: Average power consumption and memory latency reduction achieved by the TABSBC related to the baseline using **2** applications. Each bar shows a breakdown of the accesses, either satisfied in the L2 cache or in main memory.

5.5.1. Average memory latency and power consumption

The percentage that second hits represent in terms of power consumption and latency has been counted in Figures 5.9 and 5.10, which show the percentage of power consumption and latency reduction achieved by the TABSBC using 2 and 4 applications, respectively. Each bar is broken down in the percentage of hits that are satisfied in the second level of the memory hierarchy or in the main memory. We do not show the percentage due to the L1 hits because it is very similar for all the approaches. As the tag check delay means only a 3% of the total power consumption per read/write access in the cache, our approach has a negligible power consumption overhead.

TABSBC gets an average power consumption and memory latency reduction of 11% and 10% for the 2-core experiments, respectively, and 14% and 11% in the 4-core ones. This way, the more cores, the larger the advantages of the application of TABSBC.

5.6. Cost

We consider here the cost of TABSBC in terms of storage. Table 5.2 calculates the storage required for a 4MB 16-way baseline cache with lines of 64B assuming

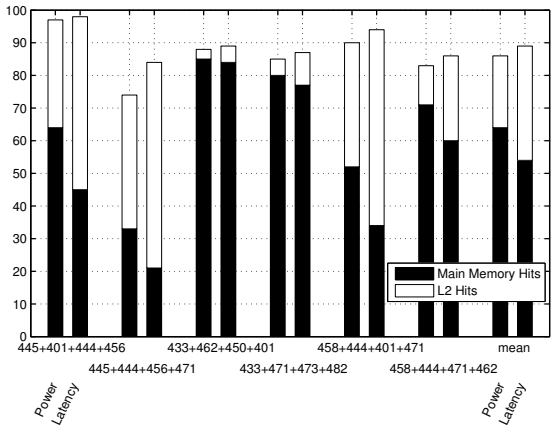


Figure 5.10: Average power consumption and memory latency reduction achieved by the TABSBC related to the baseline using **4** applications. Each bar shows a breakdown of the accesses, either satisfied in the L2 cache or in main memory.

addresses of 42 bits. TABSBC requires the following additional hardware with respect to a standard cache: a saturation counter per set to compute the SSL for each core; an additional bit per entry in the tag-array to identify displaced lines (d bit in Table 5.2); $\log_2(C)$ bits, C being the number of cores, to determine the owner core of a line (c bits in the table); an Association Table with one entry per set that stores a bit to specify whether the set is the source or the destination of the association; and the index of the set it is associated to, and, finally, a Destination Set Selector (DSS) to choose the best set for an association. TABSBC needs also one bit per set to indicate the set insertion policy for each core. Based on this, the resulting TABSBC storage overhead is about 0.65%.

Figure 5.11 shows the hardware overhead scalability of our design when varying the number of cores for three cache configurations. We can see that for typical realistic configurations the cost stays around 1% or below. In fact, unrepresentative configurations, such as a 2MB cache shared by 8 cores or a 8MB cache shared by 16, are needed to reach a 2% cost, which is still reasonable. As the cache size increases, the overhead is reduced. Also its growth is clearly sublinear with respect to the number of cores that share the cache. Altogether, while the cost of other techniques designed for shared caches is even smaller, the large performance benefits of TABSBC coupled with its small cost make it a very interesting design point,

Table 5.2: Baseline and TABSBC storage cost in a 4MB/16-way/64B/LRU cache shared between **2** cores.

	Baseline	TABSBC
Tag-store entry:		
State(v+dirty+LRU+$[d]$+$[c]$)	6 bits	8 bits
Tag ($42 - \log_2 sets - \log_2 64$)	24 bits	24 bits
Size of tag-store entry	30 bits	32 bits
Data-store entry:		
Set size	64*16*8 bits	64*16*8 bits
Additional structs per set:		
Saturation Counters	-	2*5 bits
Insertion policy bit	-	2*1 bits
Association Table	-	12+1 bits
Total of structs per set	-	25 bits
DSS (entries+registers)	-	11B
Number of tag-store entries	65536	65536
Number of data-store entries	65536	65536
Number of sets	4096	4096
Size of the tag-store	245760B	262144B
Size of the data-store	4096kB	4096kB
Size of additional structs	-	12811B
Total	**4336kB**	**4364kB** (0.64%)

particularly given the large availability of transistors and the critical role of cache performance in current systems.

5.7. Analysis

5.7.1. Scalability analysis

Table 5.3 shows the evolution of the metrics previously considered to evaluate TABSBC with respect to a baseline 16 ways cache shared by two cores for different cache sizes. These values are geometric means obtained on the 16 multiprogrammed workloads in Table 5.1. As expected the values tend to diminish with the cache size, but the reduction is not continuous or pronounced.

Figure 5.12 shows the percentage of throughput improvement with respect to the baseline 4MB 16-way cache using four cores. All the workloads used, identified by the benchmark numbers, have also an MPKI (in parenthesis) greater than 1. On average, BSBC gets an improvement of 1.8%, PIPP 1.9%, pseudoLIFO 1.6%,

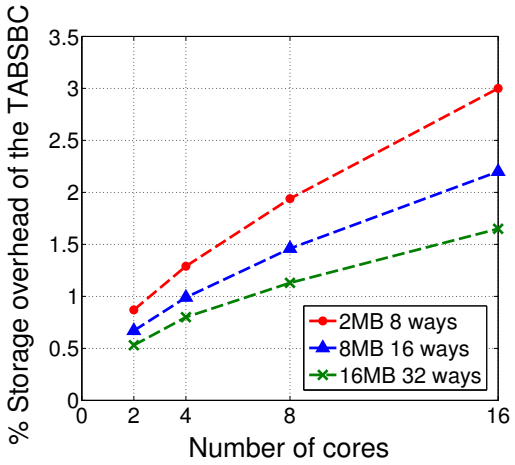


Figure 5.11: Storage overhead of the TABSBC as a function of the number of cores for several cache configurations.

Table 5.3: Percentage of throughput, weighted speedup, harmonic IPC improvement and miss rate reduction of TABSBC over the baseline, varying the cache size.

Cache size	Throughput improvement	Weighted Speedup	Harmonic IPC improvement	Miss rate reduction
1MB	4.5%	6.3%	6.3%	11.0%
2MB	5.1%	4.2%	4.3%	14.3%
4MB	4.0%	4.0%	4.4%	12.0%
8MB	3.0%	3.2%	3.5%	11.0%

TADIP 3.5%, TADRRIP 3.4% and TABSBC 5.1%. The increase in the number of cores that share the cache leads thread-aware techniques to outperform more clearly non-thread-aware ones. The exception is PIPP because the increase in the pressure that the four cores exert on this cache coupled with PIPP slow promotion policy, lead to much contention in the bottom part of the LRU stack of each set, where this technique inserts incoming lines. It is also interesting that while in two core experiments TADRRIP performed consistently better than TADIP, there is a match between them here. As for TABSBC, the variety of policies it uses, coupled with the accuracy with which it applies them to the appropriate lines and sets depending on their state in a thread-aware way, allows it to adapt very well to an increasing number of cores. Its fine granularity and accuracy are also reflected in the fact that it does not slow down any workload either in this environment. Finally, Figure 5.12

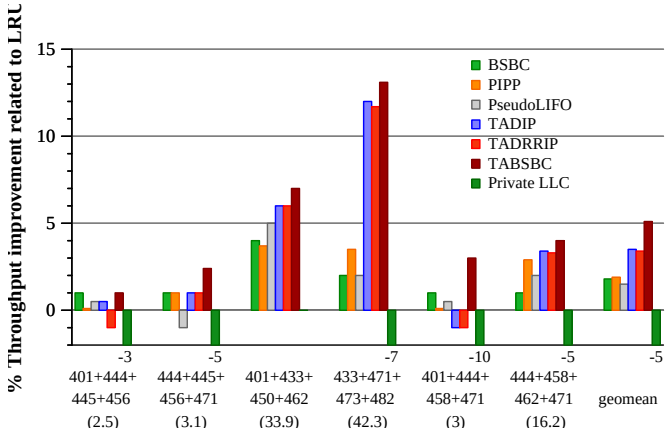


Figure 5.12: Throughput improvement over the 4MB 16 ways baseline cache shared by **4** cores using several policies. MPKI of each workload in parenthesis under its name.

includes a comparison with a CMP architecture which uses a private LLC (16 ways, 1MB per core, 8 cycles for the round trip latency and 4 for the tag check delay) per core. We can see how important the sharing of resources is, as this configuration clearly performs worse, about 5% on average, than our baseline LLC shared by the four cores.

Table 5.4 completes our scalability study with the average improvements of the six policies for the four indicators considered. TABSBC advantage over the other techniques grows with the number of cores, improving all the indicators at least about 50% more than any other strategy. The $3x$ reduction of misses it achieves with respect to BSBC is a clear indicator of the importance of thread awareness.

Table 5.4: Throughput, weighted speedup, harmonic IPC improvement and miss rate reduction of several techniques over the 4MB 16-ways baseline shared by **4** cores.

Policy	Throughput improvement	Weighted Speedup	Harmonic IPC improvement	Miss rate reduction
BSBC	1.8%	1.8%	1.8%	4.9%
PseudoLIFO	1.6%	2.0%	1.6%	4.5%
PIPP	1.9%	1.8%	1.8%	5.0%
TADIP	3.5%	3.0%	3.4%	10.0%
TADRRIP	3.4%	3.0%	3.3%	9.9%
TABSBC	5.0%	4.8%	5.6%	15.0%

5.7.2. Interaction with Prefetching

We have performed some experiments adding a simple 16KB stride prefetcher to the L2 shared cache for both TABSBC and the baseline configurations. In the 2-core experiments TABSBC got an average IPC improvement of 3.97% over the baseline, quite similar to the 4% obtained without using prefetching. As for the 4-core applications, TABSBC achieved a 6.1% improvement, which is 20% greater than the 5.1% reached in the experiments without prefetcher. According to these results, we can infer that TABSBC takes a meaningful advantage of prefetching as the number of applications sharing the cache increases. The particular management of each core that TABSBC proposes allows the prefetcher to obtain a more accurate prediction, achieving higher hit rates than the prefetcher applied to the baseline configuration. For example, as the BIP-C policy may prevent recently inserted lines from being evicted by a different core from the one that brought the line to the cache, the stream of references that the prefetcher receives has fewer interferences [83] between cores than the prefetcher used in the baseline configuration. Even more, as the prefetcher is able to perform better predictions, the TABSBC manages lines with more locality in the cache, thus achieving better results. Note that we are using a simple prefetcher, without regarding the interaction between several cores in the shared cache.

5.8. Summary

The traditional conflict and capacity misses present in single core environments, which the novel metric called Set Saturation Level (SSL) has been successfully proved to detect in the previous chapters, are found in shared caches as well. In fact, new misses of both kinds appear in shared caches due to the effects of the joint working set of the applications sharing them. From this analysis, this chapter proposes, in a reasoned way, a coordinated thread-aware strategy to reduce both capacity and conflict misses. This technique, called Thread-Aware Bimodal Set Balancing Cache (TABSBC) measures the degree of pressure that each application applies to each cache using the Set Saturation Level. When TABSBC estimates an application is experiencing problems in a set, it first tries to displace lines of

the problematic application to underutilized sets applying the Set Balancing Cache techniques. When this fails or is not possible, it resorts to a modified Bimodal Insertion Policy, which we have called BIP-C, for the problematic application, which has proven to be suitable to reduce capacity misses. Despite its fine-grained nature both in terms of measurement and modification of the cache behavior, and the variety of policies that it can apply, TABSBC cost is very reasonable; around 1% or less in representative configurations.

Large simulations using a wide range of workloads indicate four things. First, thread-awareness is a very desirable property for policies oriented to shared caches and can be successfully managed by the SSL as well. Second, among them, TABSBC achieves the best results overall consistently. This is due to two key characteristics that distinguish it clearly from the all other thread-aware techniques we are aware of. The first one is the small granularity at which it can take and apply decisions, as opposed to the global decisions of other approaches. The second one is its coordinated approach to reduce conflict and capacity misses that can balance load among sets. This latter issue is largely ignored by the other proposals specifically designed for shared caches, which only focus on the workload inside each set. We think that it is of the upmost importance to identify the dead blocks in the cache and make the best usage of them. This requires changes to the placement policy like the ones explored here. The third observation is that for the same reasons TABSBC scales very well as the number of cores that share the cache increases. Finally, it is worth to point out that the proposed BIP-C insertion policy implies a notable improvement with respect to the original BIP [53] in shared caches, as it protects lines from early displacements due to other threads.

Chapter 6

Adaptive Set-Granular Cooperative Caching

6.1. Introduction

Choosing either a private or shared configuration for the last-level cache (LLC) is one of the key points of the design in CMPs. When the LLC is shared among all the cores, it requires a high bandwidth because every single request by any upper cache needs to access the interconnection network. Shared LLCs are usually distributed in tiles owned by different cores and, thus, needing different latencies depending on where the requested line is found. As the number of cores and cache banks increases, it becomes more difficult to hide wire delays. Even worse, harmful applications can hurt the performance of other concurrently executing applications. On the other hand, in private configurations each core is assigned a static portion of the LLC, which provides lower latency, better scalability, isolation and makes easier the optimization of particular parameters like power consumption, at the cost of depriving the system of the ability of sharing underutilized resources. CMPs provide room for improving performance since multiple different applications may be executed concurrently, ones being short of cache resources while others can offer underutilized space. Therefore, it is interesting to track the global availability of resources and select the best policies to allocate them appropriately.

In this chapter we propose an approach to share resources between caches in a per-set level basis called Adaptive Set-Granular Cooperative Caching (ASCC). This design measures the degree of stress of each set and spills lines from those sets with a high number of misses related to hits, which are thus unable to hold their working set, to sets with underutilized lines in another cache, where they can be found later using the coherence mechanism. Also, ASCC adds a third neutral state where the set is disabled for both spilling or receiving lines. This way each set applies the policy which best suits it.

Another novel idea explored by ASCC is that the metrics used to manage the sharing of resources can also drive changes in the local management of a set. The fact that the same metric drives both kinds of optimization eases their desirable coordination. ASCC first resorts to spills to alleviate high miss rates. If this does not suffice, the cache may be dealing with capacity problems. ASCC adopts the innovative feature of changing the insertion policy to one specially designed to deal with capacity problems when spilling is not enough to hold the working set in the CMP.

Furthermore, experiments using different granularities for the ASCC indicate that granularity is an important point of the design. We propose an extension that dynamically adjusts the granularity of ASCC by virtually grouping adjacent sets for the sake of the tracking of their state and the application of the ASCC policies. We have called it Adaptive Variable-Granularity Cooperative Caching. Finally, we have improved this design adding support for Quality of Service.

6.2. Background and Motivation

Novel approaches, many of which are not suitable for the traditional multiprocessors, have been proposed to exploit the new capabilities of CMPs. A great number of them has focused on the memory hierarchy due to its large impact in the overall performance. By using private LLCs, a system is able to provide the applications with isolation, low latency and minimum bandwidth. Also, it allows easier design extensions, as the tags are associated to each cache. Several approaches have appeared in the past years in order to provide private levels with shared capacity.

Some techniques use partitioning in order to limit the amount of space for private and shared data, while others try to make a better use of resources by spilling lines between caches. There are also mixed approaches like the Elastic Cooperative Caching (ECC) [19], which devotes some lines of each set to allocate lines evicted from the upper level and the rest to hold lines spilled by neighbor caches. As we have previously mentioned, the main problem with the approaches that rely on partitioning [4][8][71] is that they can waste space if the allocated ways in the shared or in the private region are not useful. These approaches often waste space by forcing to allocate at least one way for each type of data, even if it is not profitable, and they need large structures to operate. Additionally, the uneven demands experienced by different sets [55][60][61][85] may result in some ways being wasted in the regions of some sets, while the same regions could benefit from more ways in other sets.

As for the designs that rely on spills, Cooperative Caching (CC) [7] disregards whether the spilling is going to benefit the cache or not. In a similar way, any cache can play the role of receiver even if it has no free space to share and the final candidate is chosen randomly. Dynamic Spill-Receive [52] (DSR) applies set dueling to label each cache either as spiller or receiver depending on a global counter per cache. Every single cache may update this global counter in order to determine whether the spillings are going to hurt receiver caches or not. A common limitation of CC and DSR is that they apply uniformly the same policy to all the cache sets, as they cannot detect whether a given set is going to perform better with more ways or applying a different policy. Even worse, all sets have to work always as spillers or receivers when sometimes it may be better to neither spill nor receive spilled lines. For example, a certain cache playing the role of spiller for the only reason it was not working well as receiver, and whose working set tightly fits in it, may spill lines to other caches requiring higher latencies to found them later and, consequently, degrading the overall performance. Furthermore, DSR restricts the number of spiller or receiver sets because they could be members of a Set Dueling Monitor (SDM) which uses a fixed policy even when a different one is performing better in the cache. In summary, there are no approaches that use a fine-grain metric to profile the state of the caches and apply different policies to different portions of the cache depending on their status. Also, no approach allows a given set to be in a neutral state, not operating as spiller or receiver. Moreover, these approaches are not designed to deal with capacity problems. Our approach, called Adaptive Set-Granular Cooperative

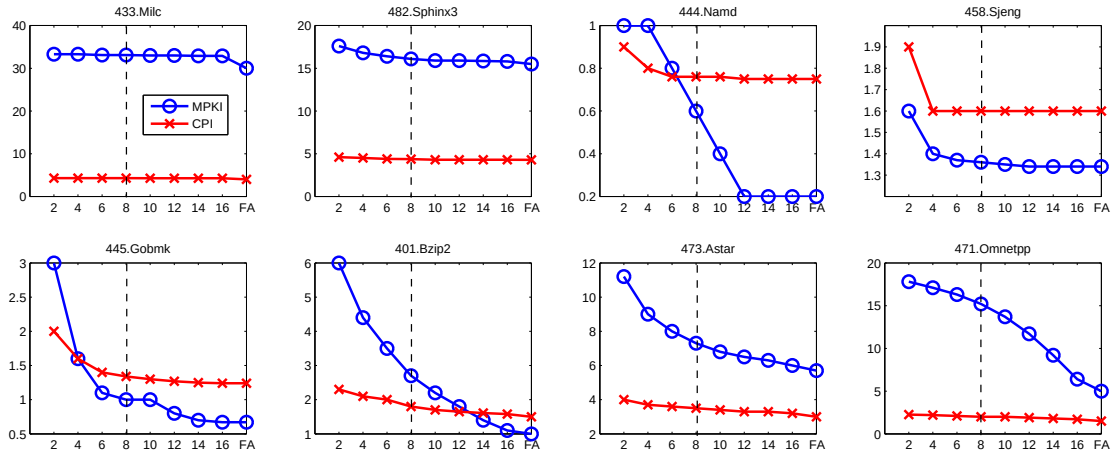


Figure 6.1: MPKI and CPI for SPEC CPU2006 benchmarks as the number of allocated ways varies. The X axis shows the number of ways allocated from a 16-way 2MB cache (the remaining ways are disabled). Benchmarks in the upper row can provide cache capacity and benchmarks in the lower row can benefit from allocating more ways gradually.

Caching (ASCC), is able to determine in a per-set basis whether the set should be a spiller, a receiver or neither of them while it tackles capacity misses at the same time. Furthermore, previous designs apply their policies using a static granularity to track the cache behavior when sometimes it may be harmful. We have designed an extension called Adaptive Variable-Granularity Cooperative Caching (AVGCC), which dynamically changes the granularity of the ASCC policies.

Figure 6.1 shows the MPKI and CPI obtained in a 2MB 16 ways L2 cache enabling from 2 to 16 ways for 8 benchmarks from the SPEC CPU2006 suite. In our experiments in Section 6.6 we have used a 1MB 8 ways L2 cache as the baseline configuration, which is represented in the graphs with a dotted line. Statistics are gathered for the first 10 billion instructions executed after the initialization. We can see how benchmarks in the lower row significantly benefit from allocating more ways, while the ones in the upper row do not. In the first row we can deduce that *milc* and *sphinx3* are streaming applications because they have a high MPKI and they are barely affected by the increase in the number of allocated ways, *namd* has a small working set and *sjeng* is sensitive to cache capacity only up to $1/4th$ MB. All of them can offer cache capacity by increasing the number of allocated ways for the

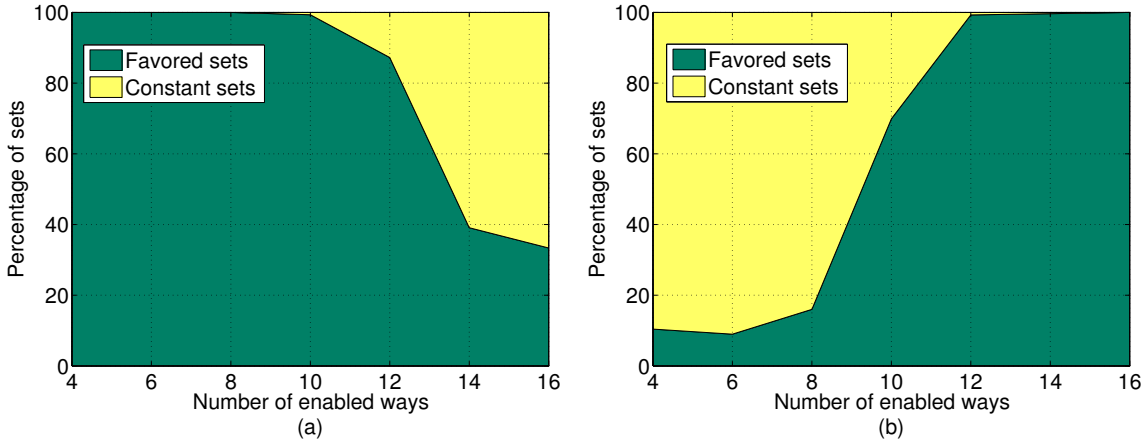


Figure 6.2: Percentage of sets that benefit from allocating more ways (favored) and percentage of sets which remain unchanged (constant) in the execution of the *Astar* (a) and *Milc* (b) benchmarks.

hungry applications. On the other hand, benchmarks in the lower row significantly benefit from allocating more ways. These benchmarks that are sensitive to cache capacity may take advantage of the simultaneous execution with applications that do not benefit from receiving extra space to hold their working sets if we provide a mechanism to reassign underutilized resources between applications. The last column in each graph shows the MPKI and CPI using full associativity in the cache. We can see how in many benchmarks there is still room for improving performance by reducing capacity misses.

Memory references are known to be non uniformly distributed across the sets of a set-associative cache [55][60][61][85]. Thus, there may be sets which might benefit from allocating a greater number of ways and sets that are able to hold their working set with their current assigned ways. This fact is illustrated by Figure 6.2, which shows the percentage of sets which benefit from enabling more ways and the percentage of sets which remain unaffected by this increase during the execution of the applications *astar* and *milc* of the SPEC CPU 2006 suite in subfigures (a) and (b), respectively. The simulation environment is the same as in the previous study. The classification is based on the MPKI of each set. If the MPKI does not decrease when the number of allocated ways increases, or if it decreases less than 1% related to the previous MPKI, calculated using 2 fewer ways, we mark this set as a constant set. Otherwise, it is a favored set. We can see how the percentage of sets, either

favored or constant, changes considerably from 10 enabled ways on in Figure 6.2 (a) and from 6 to 12 in Figure 6.2 (b).

These results indicate that, depending on the application and cache associativity, a different number of sets may benefit from getting extra ways, while others do not. Our approach tries to detect these different behaviors in the sets to perform cache-to-cache transfers, from a cache set, which is not currently able to hold its working set and would benefit from more space, to another cache where the set is underutilized. Relatedly, we can see how having a neutral state can be beneficial, that is, it may be the case that the best for a set is to neither spill to nor receive lines from another set. For instance, increasing the number of ways in Figure 6.2 (a) from 10 to 12 leaves around 90% of favored sets, while 10% remain unaffected. When we increase the number of allocated ways up to 14, only 36% of the sets whose behavior improved when going from 10 to 12 lines keep taking advantage of having more ways. The other 64% has reached its optimum behavior using 12 ways and they do not benefit from getting more ways, while allocating fewer than 12 could be harmful. As a result, a neutral state, where the set is neither spiller nor receiver, is beneficial for this latter group of sets.

Furthermore, as we could see in Figure 6.1, a noticeable percentage of the miss rate is due to capacity (and compulsory) misses. Since spills may not be enough to alleviate capacity problems, our approach tackles these problems changing the insertion policy of the sets as well.

6.3. Adaptive Set-Granular Cooperative Caching

We propose the Adaptive Set-Granular Cooperative Caching architecture to promote a better distribution of cache resources in CMP platforms that use private last levels for the cache memory hierarchy. Our proposal achieves this by spilling lines from those caches which are short of space to other ones with underutilized resources and by adapting the insertion policy of the cache sets to their demand. Our approach tries to balance the storage of the working sets of each core between caches and to reduce capacity misses by displacing lines and applying a new insertion policy, respectively. In order to track the status of each set, our proposal uses, initially,

one saturation counter per set with the same design and characteristics of the ones used in the previous chapters.

Our design classifies a set in one of three groups depending on its SSL. We have used thresholds for the different values of SSLs regarding the results presented in Chapter 2. Thus, when the SSL is below K, the high recent proportion of hits indicates that the set can hold quite successfully its working set. In this situation it is likely there are underutilized lines in the set that could be used to store part of the working set of the set with the same index in other private caches. Therefore the set is classified as a receiver set. When $K \leq SSL < 2 * K - 1$, the set has some recent hits but given the degree of pressure on it, it might be unwise to devote lines of it to store lines of the working set of other sets. This way, the set is in a neutral state where it is neither a sender nor a receiver. Finally, when the saturation counter of a set reaches its maximum value, $2 * K - 1$, the high proportion of misses related to hits indicates that the set is not able to hold its working set and is thus classified as a sender. If the line to be evicted during a replacement operation in a sender set is the last copy in the chip, our proposal tries to optimize the usage of the caching resources by spilling it to a receiver set (with the same index) in another private cache in the same level instead of evicting it to a lower level. If there are several potential receiver sets, the one with the lowest value will be selected. If a tie occurs among several caches, the destination cache is selected randomly among the ones with the lowest value. This is the only point of the design which requires further access to the interconnection network. In order to scale the design an intermediate structure per cache similar to the Spill Allocator proposed in [19] can be easily adapted. It would only require one entry per set and it would store the saturation counter value, which must be lower than K or K when there is no valid candidate, and the index of the current candidate cache. It should be updated with every miss in the other caches. Note that the low SSL in the receiver set favors that a previously spilled line is not likely to be spilled again in the near future, preventing inactive lines from being spilled repeatedly. As the spilling of lines is performed after a miss, the search of a candidate cache can be done simultaneously with the line search operation provided by the coherence mechanism, just as in [52].

6.3.1. Spilling-Aware BIP

Another novelty of our approach is that it changes a basic policy of the private caches in CMPs in response to the feedback of the cooperation mechanism. Namely, when a spiller set is not able to find a candidate receiver set, this indicates that spilling is not possible because the set has a high SSL in all caches, giving rise to a global problem of capacity. Thus, in this situation our design changes the insertion policy of the spiller set in order to avoid capacity misses. The insertion policy reverts to the traditional MRU (Most Recently Used) one when the value of the saturation counter falls below K, indicating that the capacity problem has disappeared. The Bimodal Insertion Policy or BIP [53], which inserts new lines in the MRU position of the recency stack with a low probability, ε, while it inserts most lines in the LRU position, proved to be very effective to provide thrashing protection and thus reduce capacity misses. Our design uses a variation of BIP which inserts most lines not in the LRU position, but in the previous one in the recency stack, LRU-1, in order to discard temporary data. We have called this insertion policy Spilling-Aware BIP or SABIP.

Note that using the original BIP two harmful behaviors could happen. Firstly, destination sets working under BIP could evict just inserted lines, depriving them of a chance to be reused and consequently promoted to the MRU position, not only due to local misses, but also to make room for a spilled line. Our proposal avoids that behavior by applying the restriction on the SSL value of the destination sets so that destination sets always apply MRU insertion. But this situation could also happen in sets which were previously in BIP mode and which, due to a recent good behavior, become destination sets. Here only SABIP protects the most recently inserted line, which has probably good locality given the change of behavior of the set, from being evicted due to a spill from another set. Also, as SABIP gives more chances than BIP to lines to be reused before their eviction, it generates fewer spillings of lines with some locality, as they are retained more effectively in the set.

Our approach uses an *insertion policy bit* per set to determine the current insertion policy of the set. Figure 6.3 explains the behavior of different insertion policies. Also, our approach adds swapping of lines between caches when both the requested line found in another cache and the victim line selected by the replacement policy

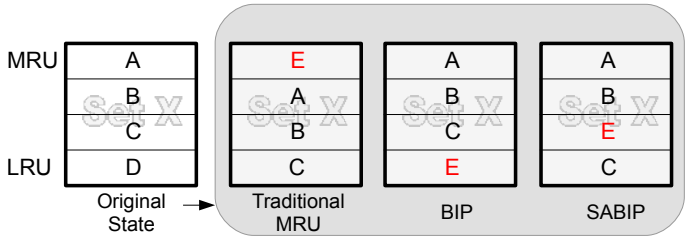


Figure 6.3: Different insertion policies for the new line E in set X in a 4-way cache.

in the cache that performed the request, are the last copy on chip. This is done in order to keep these lines longer in the CMP.

6.3.2. Design breakdown

Figure 6.4 reasserts our design decisions and measures the contribution to performance of each one of them by showing some intermediate points of the design. The experiments consider 4 cores and multiprogrammed workloads using benchmarks from the SPEC CPU2006 suite. The characterization of these benchmarks and the simulation environment will be described in Section 6.5. In this figure, LRS or Local Random Spilling is ASCC without insertion policy modifications to tackle capacity misses and choosing randomly any cache with a value in the saturation counter of the current set lower than K as a candidate to receive a spilled line. LMS or Local Minimum Spilling selects the cache with the lowest value instead. GMS or Global Minimum Spilling uses only one counter to globally manage each cache (4 bits to represent the only saturation counter per cache with an associativity of 8), so that all the cache sets have the same behavior. LMS+BIP adds BIP to LMS and GMS+SABIP (with an extra bit to determine the current insertion policy in the cache) adds SABIP to GMS. Also, we show the Dynamic Spill-Receive (DSR) [52] performance. Note that DSR is similar to GMS but using the set dueling mechanism instead of the SSL one, and ASCC is identical to LMS+BIP but using SABIP. We can see that LMS outperforms LRS thanks to the selection of the receiver cache with the lowest value for the saturation counter of the current set. In a similar way, LMS outperforms GMS thanks to its ability to handle each cache set separately. The benefits of SABIP, with respect to BIP in this environment, can be deduced comparing

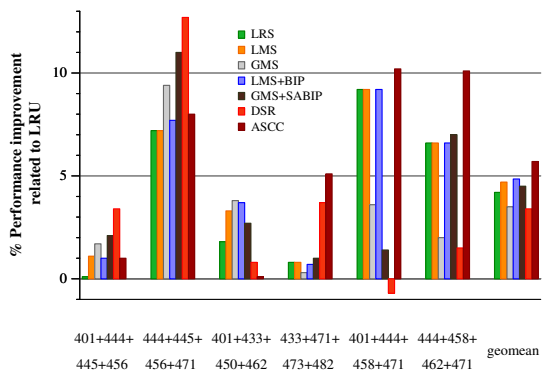


Figure 6.4: Performance improvement over the baseline for several intermediate designs of the ASCC using local (L) or global (G) policies to make decisions on spillings, for DSR and for ASCC itself running four applications.

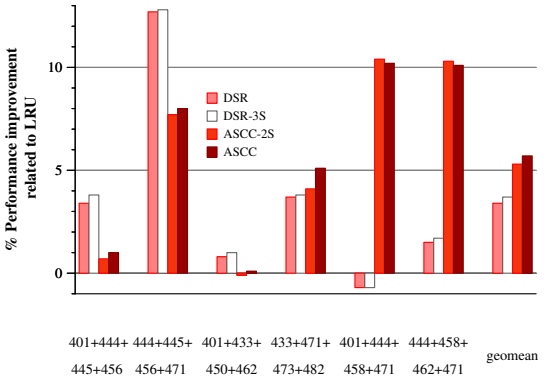


Figure 6.5: Performance improvement over the baseline for ASCC, a variation that uses only 2 states to classify the role of the sets, DSR and a variation that uses 3 states to classify the role of the cache, running four applications.

ASCC with LMS+BIP. It is worthy to point out the improvement of GMS+SABIP over DSR. This design, which uses half the negligible storage overhead of DSR, provides 30% more speedup over the baseline thanks to its management of the insertion policy. Furthermore, the comparison of GMS+SABIP with ASCC gives an idea of the value of a fined grain granularity in the cache management.

Figure 6.5 evaluates the usefulness of the neutral state of a set, in which it is neither a spiller nor a receiver. DSR with three states (DSR-3S) is a variation of DSR which uses the 2 most significant bits of its selector counter to decide if the cache is a spiller, when the 2 MSBs are 11, receiver, 00, or neutral, 01 or 10. This design achieves 9% more performance improvement over the baseline cache than the original DSR. Also, we have tried an ASCC design which uses only 2 states, ASCC-2S, in which a set is a spiller if its SSL is $>= K$ and a receiver otherwise, to see the effect of the neutral state in ASCC. We can see in the last column, which represents the geometric mean, that its performance improvement over the baseline is 10% smaller than that of ASCC.

Table 6.1: Percentage of performance improvement of the ASCC from using 4096 counters (the original ASCC design, using 1 counter per set) to using only 1 counter (grouping all 4096 sets in the cache), related to the baseline configuration.

MW	ASCC	ASCC1024	ASCC256	ASCC64	ASCC16	ASCC4	ASCC1
MW41	1%	1%	1%	1%	1%	1.2%	2.1%
MW42	8%	13.6%	15.6%	17.9%	17.9%	16.4%	11%
MW43	0.1%	0.96%	1.4%	1.4%	1.4%	1.5%	2.7%
MW44	5.1%	6.17%	5.6%	6.4%	6.2%	6.1%	1%
MW45	10.2%	5.13%	6.6%	9%	9.1%	7.1%	1.4%
MW46	10.1%	5.45%	7.9%	9.2%	9.1%	7.6%	7%
geomean	**+5.7%**	**+5.2%**	**+6.2%**	**+6.9%**	**+6.8%**	**+6.5%**	**+4.5%**

6.4. Adaptive Variable-Granularity Cooperative Caching

In Figure 6.4 we could see how the designs that apply a global metric to trigger the spilling of lines perform better in the first two of the six multiprogrammed workloads. We have performed experiments applying different granularities in ASCC to study their influence on the performance. Applying a granularity for n consecutive sets using saturation counters simply involves updating a single counter and using its value to take the decisions for the whole group. Table 6.1 shows the percentage of performance improvement over the baseline when applying ASCC grouping sets 1 by 1, 4 by 4 and so on, up to 4096, i.e, from using 4096, 1024 counters and so on, to using only one counter for the whole cache. Note that the insertion policy relies on the saturation counter, so all the sets associated to one counter apply the same insertion policy. This is sensible, as if the number of counters is small, it means we are using a global metric and the insertion policy should be global as well. Figure 6.6 shows an example for a 4-set cache applying different levels of granularity using SSLs.

From these results we can infer that some workloads, or individual benchmarks within a workload, are better managed using a global metric, while others work better with a fine granularity of management. Increasing the granularity can help to correct ASCC overreactions to temporary outstanding behaviors of particular sets, while decreasing it allows to track the state of the cache sets and detect capacity problems in a more accurate way. Therefore, we have two options. The first one

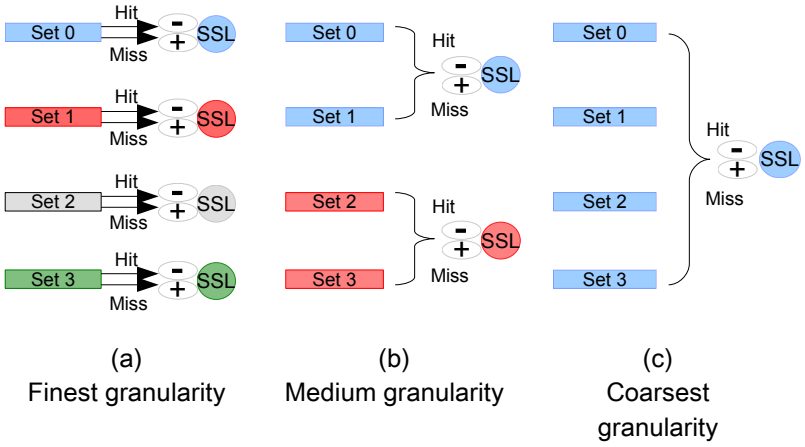


Figure 6.6: Different levels of granularity.

is to fix statically the granularity to some value that seems to work well overall. The drawbacks of this option are that it is not optimal, it does not adapt to the nature and the changes of behavior of the workloads, and there is no guarantee the granularity selected will be far from good for some workloads. The second option is to design a dynamic mechanism to adapt the algorithm granularity to the needs of the applications during the execution. Our proposal entails starting with one counter for the whole cache. The number of counters used is increased, duplicating the current number, if more than half the saturation counters in use have a value lower than K. When this happens, it means that most sets in the cache can provide sets in other caches with space, so we use a finer granularity to allow the sets to exchange lines in a more accurate way. Also, the current number of counters in use is decreased, halving it, when every pair of neighbor counters has a similar value, specifically when there is an absolute difference between their values of two at the most, and their embraced sets are working under the same insertion policy. This is sensible, since we can save storage space using one counter to track their SSL simultaneously, as they are similar. Note that different caches in the same CMP can be applying different granularities. We have called this technique Adaptive Variable-Granularity Cooperative Caching or AVGCC.

6.4.1. Hardware description

In order to implement the described mechanism, AVGCC needs three counters per cache. The first one, D henceforth, stores the logarithm of the current number of sets per saturation counter in use to base 2. That is, D is the logarithm to base 2 of the granularity applied by AVGCC. The saturation counters are then accessed adding a shifter controlled by D in the indexing path. This way, given a set index I, the index of the associated counter would be I >> D, where the $a >> b$ operator represents an arithmetic shift of b bits to the right of integer a.

Secondly, AVGCC uses another counter, A, in order to track how many pairs of SSLs fulfill the conditions required when checking whether the number of counters in use must be halved. The condition is evaluated twice, before updating the accessed SSL, and after doing it. A flip-flop is needed to check whether the evaluation of the condition between the given SSL and its adjacent one has changed after updating the former. Counter A is decreased if the evaluation of the condition turns from being fulfilled to not being, increased in the opposite case, and it remains unchanged otherwise. Finally, the number of counters in use must be duplicated if more than half the saturation counters have a low value (below K). A counter, B, is needed for this purpose. B is increased when the value of a saturation counter goes from K to K-1 and decreased when it changes from K-1 to K. Furthermore, as it was previously mentioned, the D counter is increased when $A = (S >> D)/2$, where S is the number of sets, as every pair of the $(S >> D)$ saturation counters in use fulfills the condition for it. Counter D is decreased when $B > (S >> D)/2$. Also, after updating the current number of counters, the new ones are initialized to $K - 1$ and the associated insertion policies are reset to the traditional MRU one. This process is performed periodically. We will see in Section 6.6 that this design provided an average improvement of 7.8% over the baseline for the same set of workloads and configuration used in Table 6.1 in comparison with the 6.9% of the best static approach.

Table 6.2: Architecture of the CMP with private LLCs.

Processor	
Frequency	4GHz
Fetch/Issue	4/4
ROB entries	176
Integer/FP registers	96/80
Memory subsystem	
L1 i-cache & d-cache	32kB/2-4-ways/32B/LRU/WT
L2 (unified, inclusive) cache	1MB/8-way/32B/LRU/WB
L2 Cache latency (cycles)	9 **local** hits, 25 **remote** hits
Main memory latency	115ns
Coherence protocol	MESI-based broadcasting

6.5. Simulation environment

To evaluate our approach we have used the SESC simulator [58] with a baseline configuration based on four-issue out-of-order cores with two cache levels, both of which are private to each core. This configuration is detailed in Table 6.2. The ratio of LLC space per core is similar to that used in the related bibliography and actual processors [15] [1].

We have used the same 13 benchmarks of the SPEC CPU 2006 suite as in the previous chapters with an MPKI of at least 1, as shown in their characterization in Table 6.3, to make 14 multiprogrammed workloads of two applications and 6 of four. These workloads cover combinations between applications that benefit from allocating more ways and other ones that do not, workloads where no benchmark benefits from getting extra space and workloads where all the benchmarks would benefit from getting it.

Table 6.3: Benchmarks characterization.

Benchmark	L2 MPKI	CPI	Benchmark	L2 MPKI	CPI
401.bzip2	2.7	1.8	458.sjeng	1.36	1.6
429.mcf	40.1	10.4	462.libquantum	22.4	4.3
433.milc	33.1	4.28	470.lbm	29	2
444.namd	1	0.76	471.omnetpp	15.2	2
445.gobmk	1.1	1.34	473.astar	7.3	3.5
450.soplex	3.6	1	482.sphinx3	16.1	4.37
456.hmmer	3.4	1.3			

They have been executed using the reference input set (*ref*), during 10 billion instructions after the initialization. When each core reaches this number of instructions it continues its execution until the last core finishes, in order to keep competing for the cache resources.

6.6. Experimental evaluation

ASCC, as well as the other approaches, has been applied in the last level of the cache memory hierarchy for every single core. In our experiments we have also tested DSR with 32 sets per Set Dueling Monitor and 1 SDM per policy, a combination of DSR and DIP [53], where DIP decides the insertion policy for the global cache (either BIP or the traditional LRU one) depending on which policy is working better using also set dueling and, finally, the ECC approach described in Section 6.2. Our designs, as well as the DIP used in the combination with DSR, use a probability $\varepsilon = 1/32$ of inserting the new line in the MRU position using BIP. ECC uses the values proposed in [19] for the thresholds and we have implemented it without the distributed structures they propose, tracking the shared state of the lines with an additional bit per block. Note that this implementation provides the ECC design with more accuracy than the original design, which cannot track the information of all lines in the cache, specially if the degree of replication is low. Finally, AVGCC checks whether the number of counters must be changed every 100000 accesses to the cache.

Figures 6.7 and 6.9 (a) show the performance improvement over the baseline for the different approaches, measured as the weighted speedup of CPIs, using 2 and 4 cores, respectively. Numbers above or beneath the bars provide the percentages that are outside the scale range. The last column shows the geometric mean for each design. ASCC, which gets 6.4% and 5.7% of performance improvement over the baseline when executing 2 and 4 applications, respectively, and AVGCC, which achieves 7% and 7.8%, respectively, clearly outperform the other approaches. DSR adapts to the requirements of the applications thanks to the set dueling mechanism, but it is not able to take advantage of the state of each set as ASCC and AVGCC do. Also, it forces sets to be either spillers or receivers and lacks of a policy oriented to capacity problems. DSR+DIP outperforms DSR executing 2 applications

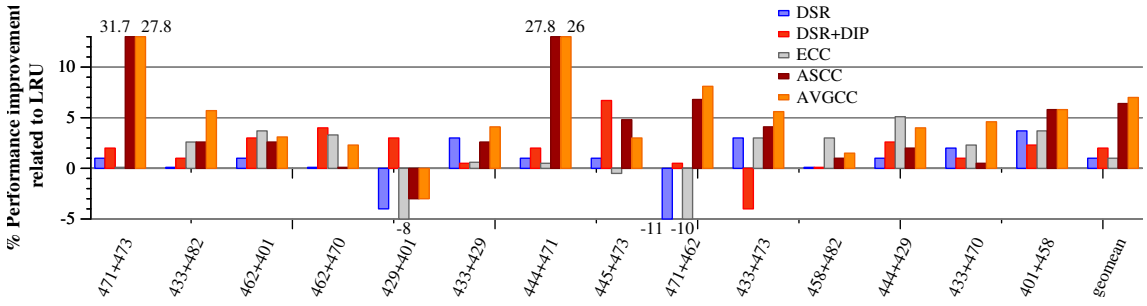


Figure 6.7: Performance improvement for DSR, DSR+DIP, ECC, ASCC and AVGCC running two applications over the baseline.

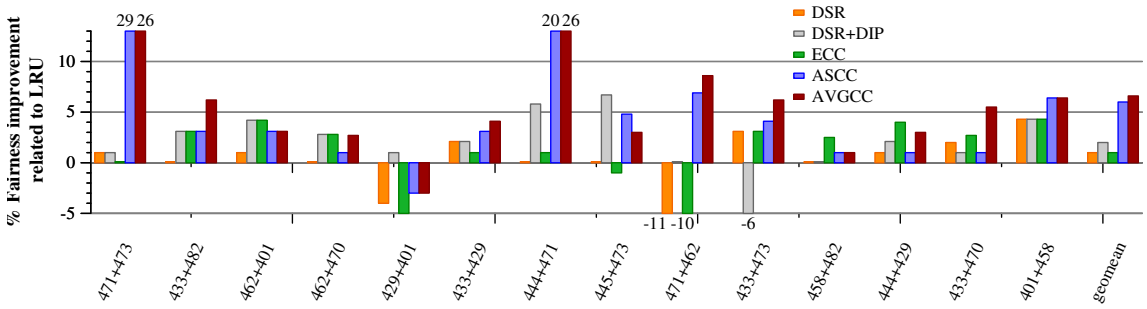


Figure 6.8: Fairness improvement for DSR, DSR+DIP, ECC, ASCC and AVGCC running two applications over the baseline.

because it tackles capacity misses as well. The problem with DSR+DIP is that its BIP insertion policy is not aware of the spillings, as we explained in Section 6.3.1. Thus, one just inserted line, which is not likely to be reused in the near future if the set is applying BIP, can be spilled to another cache and, as a result, a line with more locality could be evicted there. Even worse, a spilled line can evict a just inserted line in a set applying BIP depriving it of a chance to be reused and consequently promoted to the top of the recency stack. Let us recall that this particular behavior cannot happen in our designs. As the number of executing applications increases, the number of candidate caches to receive spilled lines increases as well, so the negative effects of BIP are likely to happen more frequently. That is why DSR+DIP degrades the performance of DSR executing 4 applications. The discrete and even negative behavior of DSR+DIP with respect to DSR emphasizes the need for modifying policies to adapt them to different environments, as we have done

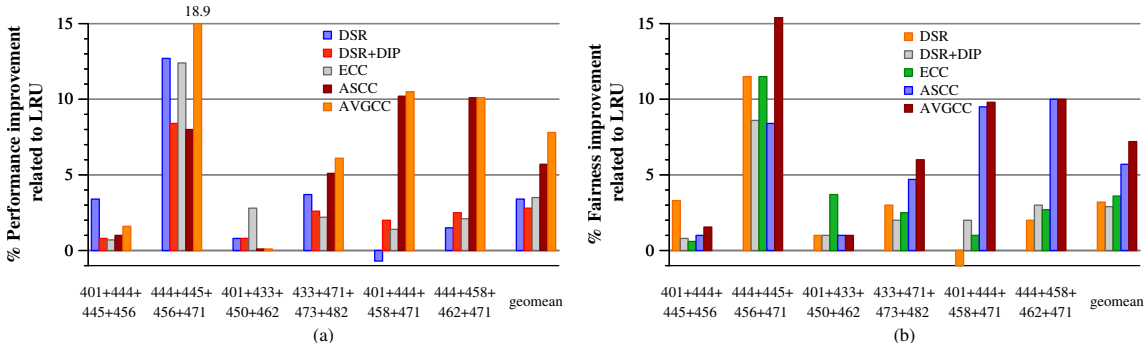


Figure 6.9: Performance (a) and fairness improvement (b) over the baseline for DSR, DSR+DIP, ECC, ASCC and AVGCC running four applications.

with SABIP, and for integrated management designs such as ASCC and AVGCC. The ECC performance improvement is modest compared to those of ASCC and AVGCC because it mainly relies in a high degree of replication, as it uses scalable structures that cannot hold information for all the lines in the cache and has the problems inherent in partitioning described in Section 6.2. Finally, AVGCC outperforms ASCC by adapting the granularity of its policies to the different requirements of the applications.

Figures 6.8 and 6.9 (b) show the percentage of fairness improvement with respect to the baseline system of each one of the considered approaches calculated as the harmonic mean of IPCs, using 2 and 4 applications, respectively. ECC gets better results than DSR and DSR+DIP because it is able to reduce the execution time for the longest applications. Overall the results and therefore the explanations for them, are similar to the ones obtained in the performance analysis. From these results we can conclude that ASCC and AVGCC do not hurt fairness when speeding up mixed workloads, AVGCC being again the leader thanks to its larger flexibility. Figures 6.10 and 6.11 perform the same study in terms of throughput improvement and show similar trends to those observed in the previous studies.

We have also simulated the usage by all the cores of a shared cache of the same aggregated capacity in which addresses are mapped to banks in an interleaved way. This cache has been simulated using an average latency (almost twice the latency of a private L2 in the baseline for the 2-core experiments and almost four times using 4 cores) for the accesses to the different banks, assuming a uniform distribution of the accesses across the banks given by the interleaved mapping. For the sake

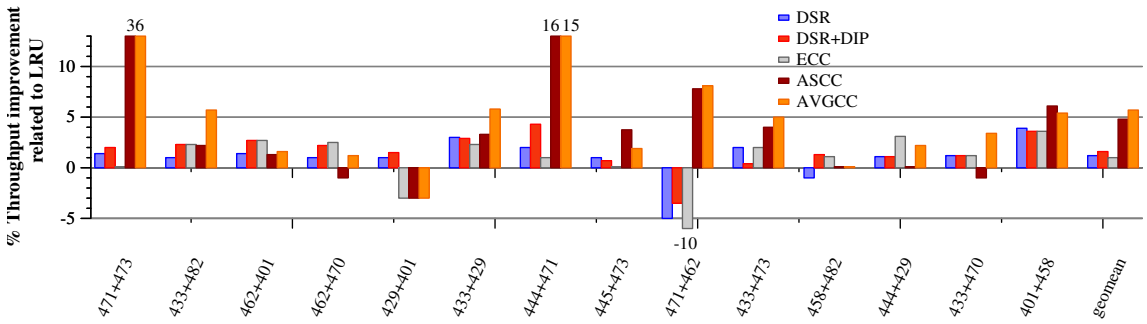


Figure 6.10: Percentage of throughput improvement for ECC, DSR, DSR+DIP, ASCC and AVGCC running two applications over the baseline.

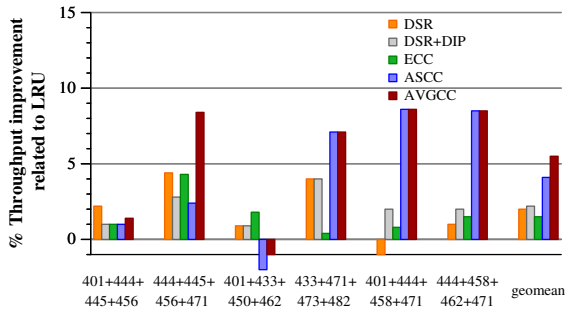


Figure 6.11: Percentage of throughput improvement for ECC, DSR, DSR+DIP, ASCC and AVGCC running four applications over the baseline.

of clarity the results for these simulations are not shown in the figures. Globally, the 2MB shared cache outperformed the baseline private configuration in the 2-core experiments by 1.8% and 1.7% in terms of performance and fairness, respectively, laying quite far from the performance of AGCC and AVGCC. Using 4 cores, the 4MB shared cache got a 3% of improvement in both metrics. This means that, in general, and despite the automatic sharing of resources inherent in shared caches, private designs with additional sharing mechanisms can be more effective by protecting sets from being assimilated by greedy cores.

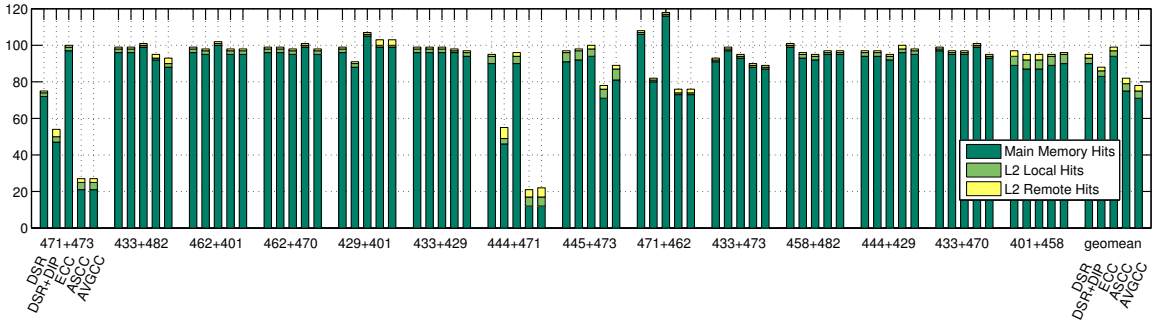


Figure 6.12: Percentage of improvement in the average memory latency showing the percentage of hits in main memory, in local L2 or in remote L2 for DSR, DSR+DIP, ECC, ASCC and AVGCC running two applications over the baseline.

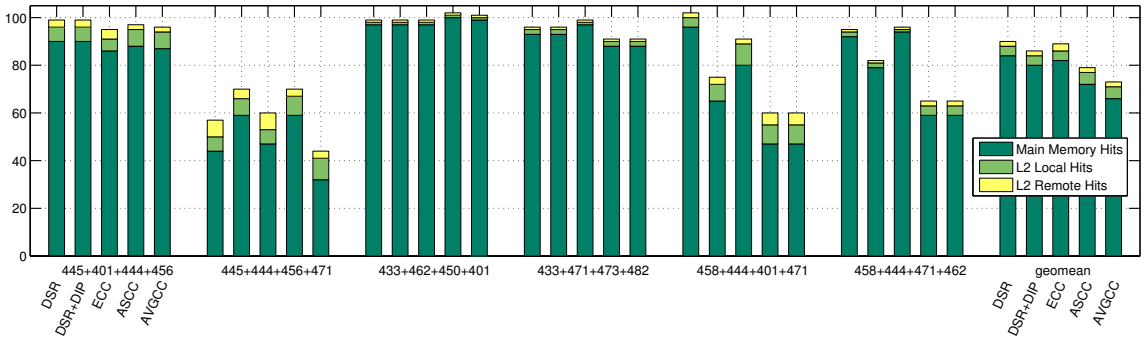


Figure 6.13: Percentage of improvement in the average memory latency showing the percentage of hits in main memory, in local L2 or in remote L2 for DSR, DSR+DIP, ECC, ASCC and AVGCC running four applications over the baseline.

6.6.1. Average memory latency and power consumption

Figure 6.12 shows the average memory latency normalized to the baseline configuration (represented by the horizontal dotted line of 100) for all the approaches in the 2-core configuration. Each bar is broken down showing the percentage of accesses to the L2 which result in hits in the local L2, in a remote one or in main memory. The percentage due to the L1 hits is not shown because it is almost the same for all the approaches. The average memory latency has been calculated regarding that each access is sequentially processed, without overlaps between accesses. This bar graph provides some feedback on previous results. For instance, we can infer that ASCC and AVGCC degrade the baseline performance for the workload 429+401 (*mcf+bzip2*) because most local L2 hits in the baseline become remote L2 hits for both approaches. The last column shows the geometric mean. DSR gets a 5% of improvement, DSR+DIP 12%, ECC 1%, ASCC 18% and AVGCC 22% in the 2-core configuration. As for the 4-core configuration, whose results are shown in Figure 6.13, DSR outperforms the baseline design by 10%, DSR+DIP by 14%, ECC by 11%, ASCC by 21% and finally AVGCC by 27%. This translates in average power consumption reductions in the memory hierarchy of 22% and 20% for ASCC and of 25% and 29% for AVGCC in our 2 and 4-core experiments, as Figures 6.14 and 6.15 show, respectively. Although DSR outperformed DSR+DIP in terms of weighted speedup and fairness (see Figure 6.9) here it is DSR+DIP the one which obtains the best results. This is due to the best performance DSR obtains in those applications with low IPCs, since as the executions take longer, the problems of DSR+DIP are exacerbated. This behavior is also asserted by the throughput analysis, which is not so dependent on the particular improvement over a single application but on the full workload, where DSR+DIP outperforms DIP (see Figure 6.11).

6.7. QoS-Aware AVGCC

In some CMPs losing performance may be unacceptable. AVGCC degrades the baseline performance sometimes. In order to solve this problem we propose an extended design to provide Quality of Service (QoS). Our design can inhibit AVGCC by stopping spillings and fixing the insertion policy to MRU. We leverage

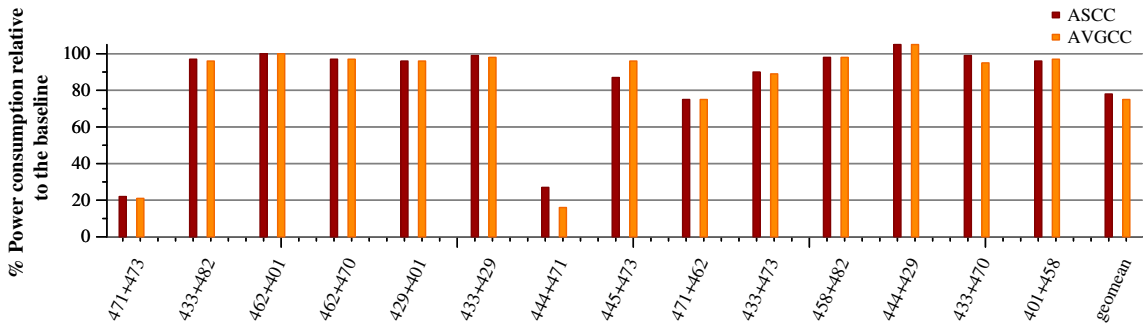


Figure 6.14: Percentage of power consumption of ASCC and AVGCC related to the baseline running two applications.

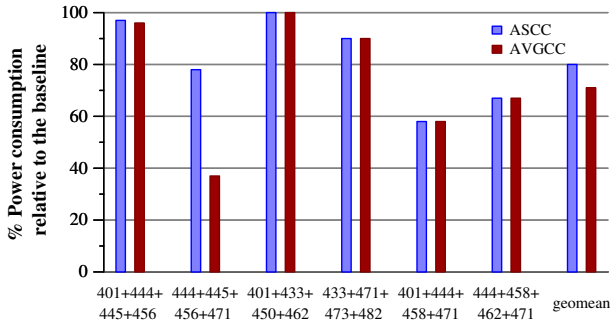


Figure 6.15: Percentage of power consumption of ASCC and AVGCC related to the baseline running four applications.

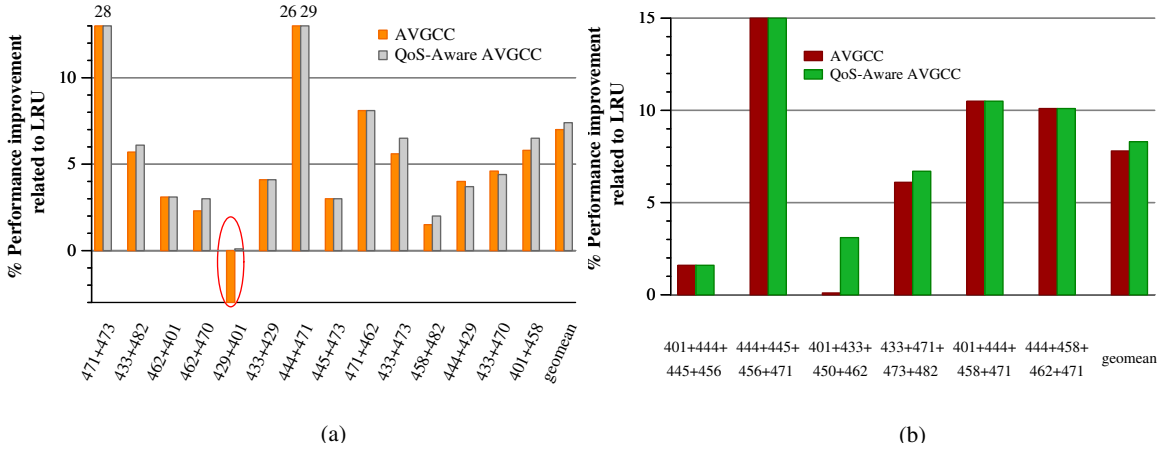


Figure 6.16: Percentage of performance improvement for QoS-Aware AVGCC and AVGCC over the baseline using **2** (a) and **4** (b) cores.

the fact that this inhibition can be done by limiting the increase that an update in the saturation counters means when a cache miss occurs. A harmful operation of AVGCC is detected when its number of misses is greater than in the baseline cache. The number of misses of the baseline cache (MBC) is estimated by tracking the misses ($SampledSetMisses$) in those sets ($SampledSets$) working under the MRU traditional insertion policy and which have a value for their saturation counters greater than $K-1$, as these sets cannot receive lines. Then MBC is estimated as:

$$MBC = CacheSets * (SampledSetMisses/SampledSets) \tag{6.1}$$

As for the number of misses for AVGCC ($MissesWithAVGCC$), it is simply collected using a counter. Our design calculates a ratio called $QoSRatio$ whose purpose is to adjust the saturation value of the sets in order to penalize or reward them depending on their behavior. It is calculated every 100000 cycles simultaneously with the recalculation of the number of counters following the equation:

$$QoSRatio = MBC/max(MBC, MissesWithAVGCC) \tag{6.2}$$

After the computation all the parameters are initialized and the saturation counters are updated after each miss by adding the $QoSRatio$, while they are decreased

in 1 unit after a hit as usual. Figure 6.16 (a) shows the percentage of performance improvement over the baseline system of the Quality of Service Aware AVGCC using two cores. Our Quality of Service approach globally outperforms the original AVGCC, as stated in the last column of the bar graph. Figure 6.16 (b) shows the same study using 4 cores, where AVGCC did not degrade the performance of any workload. In this case our QoS approach gets 8.1% improvement over the baseline.

6.8. Cost

In this section we evaluate the cost of ASCC, AVGCC and QoS-Aware AVGCC in terms of storage requirements.

ASCC and AVGCC require additional hardware because of the need of a saturation counter per set to monitor its behavior and one additional bit per set in order to determine the insertion policy. Also, AVGCC needs three additional counters: one to track the current number of counters in the cache (the D counter explained in Section 6.4), another one to determine if all pairs of neighbor sets have a similar value (the A counter) and a last one to check if there are more than half counters with a low value, lower than K, in their saturation counters (B counter). Finally, the QoS-Aware AVGCC requires a per-core storage overhead of 2 bytes for both miss counters (*SampledSetMisses* and *MissesWithAVGCC*), 4 bits to store the *QoSRatio* value (1.3 fixed point format) and 12 bits (assuming 4096 sets in the cache) to count the number of sampled sets. Also, 3 additional bits are needed per saturation counter (4.3 fixed point format). Based on this, Table 6.4 calculates the storage required for a baseline 8-way 1MB cache with lines of 32B assuming addresses of 42 bits. Altogether, the QoS-Aware AVGCC design means a 0.35% of storage overhead over the baseline considering the finest granularity, that is, having one counter per set. We can also see that the storage overhead of the other set-granular cooperative approaches means less than a 0.2% of additional storage overhead.

Table 6.4: Baseline, ASCC, AVGCC and QoS-Aware AVGCC storage cost in a 1MB/8-way/32B/LRU cache

	Baseline	ASCC	AVGCC	QoS-Aware AVGCC
Tag-store entry:				
State(MESI+LRU)	5 bits	5 bits	5 bits	5 bits
Tag ($42 - \log_2 sets - \log_2 32$)	25 bits	25 bits	25 bits	25 bits
Size of tag-store entry	30 bits	30 bits	30 bits	30 bits
Data-store entry:				
Set size	32*8*8 bits	32*8*8 bits	32*8*8 bits	32*8*8
Additional structs per set:				
Saturation Counters	-	4 bits	4 bits	4+3 bits
Insertion policy bit	-	1 bit	1 bit	1 bit
Total of structs per set	-	5 bit	5 bits	8 bits
Adjustment mechanism:				
A,B & D counters	-	-	12+12+4 bits	12+12+4 bits
Total	-	-	28 bits	28 bits
QoS-Aware mechanism:				
Miss counters	-	-	-	2+2B
QoSRatio	-	-	-	4 bits
Total	-	-	-	36 bits
Number of tag-store entries	32768	32768	32768	32768
Number of data-store entries	32768	32768	32768	32768
Number of sets	4096	4096	4096	4096
Size of the tag-store	120kB	120kB	120kB	120kB
Size of the data-store	1MB	1MB	1MB	1MB
Size of additional storage	-	2560B	2560B+ ∼4B	4104B
Total	**1144kB**	**1146kB** (0.17%)	**1146kB** (0.17%)	**1148kB** (0.35%)

Table 6.5: Percentage of performance improvement and storage overhead of AVGCC limiting the maximum number of set saturation counters from 128 to 2048 and AVGCC itself related to the baseline configuration using 4 cores.

Design	% Speedup	Additional Storage
AVGCCMax128	6.8%	83B
AVGCCMax256	6.5%	163B
AVGCCMax512	6.4%	323B
AVGCCMax1024	6.7%	643B
AVGCCMax2048	7.1%	1284B
AVGCC	7.8%	2564B

6.8.1. Limiting the maximum number of counters

As we could see in Section 6.4, it is not necessary to have one counter per set in order to get the best overall results with ASCC. Thus, we performed some experiments limiting the maximum number of counters in AVGCC in order to further reduce the storage overhead. Table 6.5 shows the performance obtained and the storage overhead over the baseline limiting the maximum number of counters from 128 to 2048 (our baseline has 4096 sets) using the multiprogrammed workloads for 4 cores. We can observe that AVGCC keeps a high degree of improvement even needing a negligible storage overhead related to the baseline.

6.9. Analysis

6.9.1. Impact of varying cache parameters

In this section we evaluate how the performance of AVGCC varies with respect to the main parameters of the cache.

Table 6.6 shows the percentage of reduction in the number of off-chip accesses, measured taking in account all the accesses from every core, achieved by the AVGCC as well as the per-core storage overhead it involves as the cache size varies between 1MB and 4MB. We can observe that the improvements are smaller from the 2MB cache on because the miss rate decreases as the cache size increases.

Table 6.7 shows the percentage of reduction in the number of off-chip accesses achieved by AVGCC, as well as the storage overhead it involves, as the associativity varies from 8 ways through 32 ways for the fixed baseline cache size. The performance improvement provided by AVGCC decreases due to two reasons. First, the increased associativity reduces the baseline miss ratio due to the elimination of conflict misses. Second, as the number of lines per set increases, the finest granularity that AVGCC can manage, a single set, is larger. Restricting the granularity hurts the performance of some workloads, as seen in Table 6.1. Still, AVGCC keeps a high degree of improvement for large associativities, while its storage overhead drops strongly.

Table 6.6: Cost-benefit analysis of AVGCC as a function of the cache size.

Cache size	% Average reduction in off-chip accesses (4 / 2 cores)	Storage Overhead
1MB	27% / 14%	0.17%
2MB	12% / 9%	0.17%
4MB	12% / 9%	0.17%

Table 6.7: Cost-benefit analysis of AVGCC as the associativity varies.

Ways	% Average reduction in off-chip accesses (4/2 cores)	Storage Overhead
8	27%/14%	0.17%
16	22%/12%	0.13%
32	20%/11%	0.08%

Table 6.8: Cost-benefit analysis of AVGCC varying the line size.

Line size	% Average reduction in off-chip accesses (4/2 cores)	Storage Overhead
32	27%/14%	0.17%
64	22%/12%	0.09%
128	18%/9%	0.05%

Table 6.8 shows the percentage of reduction in the number of off-chip accesses achieved by AVGCC, as well as the storage overhead it involves, as the line size varies from 32 bytes to 128 bytes in the entire cache memory hierarchy, remaining the other basic parameters as in the baseline cache. The average improvement obtained by AVGCC decreases as the line size increases because most of the benchmarks have a high spatial locality (7 out of the 13 benchmarks notably reduce their miss rate increasing the line size, while only 2 degrade their performance). Also, as in the case of the increase in associativity, the growth in the line size limits the minimum amount of cache resources that AVGCC can track and manage independently, which is harmful for some workloads. Still, the storage overhead of AVGCC drops much more than the improvements it brings, making it even more effective in relative terms.

6.9.2. Multithreaded experiments

We have performed experiments with multithreaded applications in order to evaluate our proposals in environments where sets tend to have a uniform demand in all caches. For these experiments we have used benchmarks from the SPLASH2 and PARSEC suites running them during 10 billion instructions (most of them

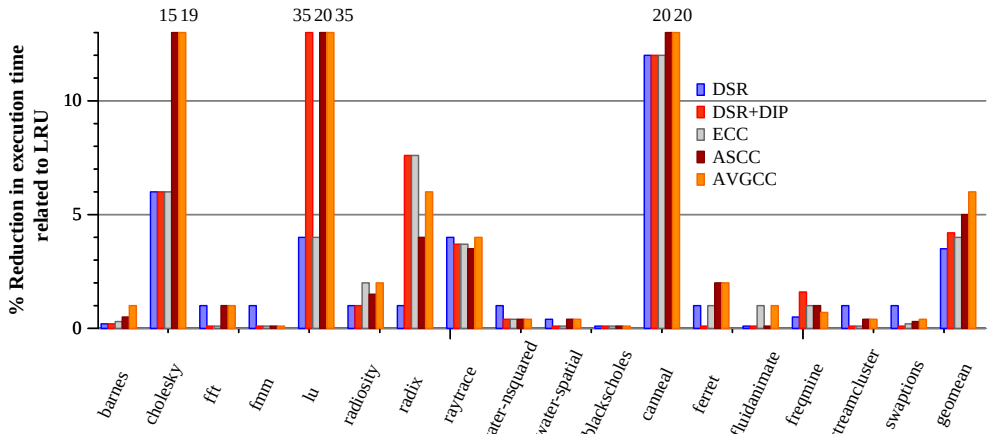


Figure 6.17: Percentage of reduction in execution time over the baseline for ECC, DSR, DSR+DIP, ASCC and AVGCC running multithreaded applications with four threads.

until completion) and using the large input set for PARSEC and the appropriate input set for SPLASH2 using 4 threads. Most of these benchmarks are not hard memory demanding so we have reduced the L2 capacity to 512KB to get meaningful results. Figure 6.17 shows the percentage of reduction in execution time over the baseline using 4 threads. We can see that ECC performs well in this environment, even though it only outperforms DSR again. AVGCC gets a 6% of throughput improvement over the baseline, ASCC a 5%, DSR+DIP 4.5%, ECC 4% and DSR 3.5%. In this environment the spilling of lines can benefit even the receiver caches, which may need the line in the near future, so our policies aim to take advantage of this behavior as well.

6.9.3. Interaction with Prefetching

Furthermore, we added a 16KB stride prefetcher to each LLC in the multiprogrammed experiments obtaining similar results. Namely, ASCC outperformed the baseline by 6% and 5.5% and AVGCC by 6.4% and 7.6% in the 2 and 4 core configurations, respectively. Prefetchers reduce the miss rate in the LLC, but this is done at the expense of consuming more bandwidth. This way, the impact of our policies is slightly reduced by the presence of prefetchers in the 2-core configuration. The reduction that the presence of the prefetchers implies in the advantages provided

by ASCC and AVGCC is negligible in the 4-core CMP because as the number of cores increases the bandwidth savings provided by these techniques are much more critical, particularly as the prefetchers consume more bandwidth.

6.9.4. AVGCC behavior

In this section measurements on the internal behavior of AVGCC are compared to those of other techniques in order to better understand how it achieves its results. We performed experiments on two parameters, the number of spillings and hits per spilled line. In the 2-core experiments AVGCC performed on average 13% fewer spills than the second best approach (DSR+DIP henceforth) and 60% fewer than the worst one (ECC). Also, its ratio of hits per spilling was 28% larger than that of the second best policy. Regarding the 4-core experiments, AVGCC performed 28% fewer spillings than the following best approach, and 70% fewer than the worst case. Also, its ratio of hits per spilling was 36% greater than the one of the DSR+DIP approach.

From these results we infer that AVGCC is not only able to achieve a higher ratio of hits per spilled line, but it also needs fewer spillings to get it. This means that AVGCC performs fewer useless spillings by relying, among other points of the design, on the neutral state of the sets.

6.10. Summary

In this chapter the Adaptive Set-Granular Cooperative Caching (ASCC), a new design aimed at last-level caches in CMP private configurations with a good cost-benefit relation, has been presented. This cache detects the different degrees of demand of the sets in each cache in order to balance their working set by spilling lines to other caches where the set is underutilized. When the sharing of resources does not suffice, ASCC modifies the cache insertion policy in order to deal with the problem of capacity. Furthermore, we propose Adaptive Variable-Granularity Cooperative Caching (AVGCC), which is able to adapt the granularity with which the sets should be profiled and managed. Also, this design has been improved with

a *Quality of Service* mechanism.

As far as we know this is the first approach that uses spills for sharing resources using a set level metric. It is also the first one able to adapt its granularity depending on the behavior of the cache and capable of coordinating a spilling mechanism with a local policy to tackle capacity misses. Finally, it demonstrates the benefits of operating portions of the cache, groups of sets in this case, neither as spillers nor as receivers of lines, that is, not participating in the spilling mechanism.

In a 4-core system running multiprogrammed workloads, AVGCC achieved a performance improvement of 7.8% with respect to the baseline. Also, it clearly outperformed recent proposals both in terms of speedup and fairness, while having a very small storage overhead. We have even proposed a model which only needs 5 bits to get a 4% of improvement. The 27% average memory latency reduction and 29% power consumption reduction it provided in these tests are also remarkable. Similar results were obtained in experiments with multithreaded applications.

Chapter 7

Conclusions and Future Work

7.1. Conclusions

In order to neutralize the gap between processor and memory speeds, modern processors rely on large last-level caches (LLC). The importance of cache management has become even more critical because of the growing bandwidth requirements of CMPs, the increasing working sets of many emerging applications, and the smaller cache size available for each core as the number of cores on a single chip continues to increase. This PhD Thesis, *"Cache Design Strategies for Efficient Adaptive Line Placement"*, analyzes some of the problems related to the management of caches and the common harmful behaviors they experience, and proposes simple and effective solutions to them that substantially improve the performance and power consumption of computers. A common characteristic of all the designs proposed in this dissertation is that they are fine-grained both in terms of analysis of the behavior of the cache and the enforcement of policies to improve its performance. This fine granularity is achieved nevertheless with very light additional hardware structures and very small changes to the standard design of caches.

This Thesis proves that the harmful effects derived from one of the inefficiencies commonly found in cache memories, the non-uniformity of the distribution of the memory references among the cache sets, can be reduced by displacing lines from cache sets which are not able to hold their particular working set to other underuti-

lized ones. The *Set Balancing Cache* (SBC) [60] associates both kinds of cache sets, in order to balance the load among them and thus reduce the miss rate, by relying on a cost-effective metric called *Set Saturation Level*, which measures the degree to which a set is able to hold its particular working set. This value, which indicates the degree of pressure on a set, is provided by a counter per set called saturation counter which is increased when a miss occurs and decreased with every hit. This metric has proved successful at setting the most suitable role, namely source or destination, of a given set in the associations. An initial static SBC design (SSBC), which only allows displacements between preestablished pairs of sets, achieved an average reduction of 9.2% of the miss rate, or 14% computed as the geometric mean. This led to average IPC improvements between 2.7% and 3.7% depending on the type of configuration tested. Furthermore, a low-cost and efficient structure, called *Destination Set Selector* (DSS), is proposed in order to provide the best set available in the cache during an association request, which yields near-optimal selections. An improved dynamic extension of the SBC (DSBC), that tries to associate each highly saturated set with the less saturated cache set available by using the DSS, obtained an average reduction of 12.8% of the miss rate, 19% computed as the geometric mean. This led to average IPC improvements between 3.5% and 5.25% depending on the type of memory hierarchy tested. Furthermore, the SBC designs proved consistently to be better than increasing the associativity, both in terms of area and performance, as well as implying negligible storage overheads, less than 0.6% related to the baseline cache.

This dissertation also confirms that another one of the problems usually found in cache memories, thrashing, which appears when lines with a certain degree of locality are evicted to make room for other temporary ones, can be alleviated by applying a suitable insertion policy. This insertion policy must be able to discard temporary data as soon as possible while retaining the most important part of the working set in the cache. This kind of policy can be combined with the DSBC design presented in this Thesis, to decrease the pressure on the cache sets when the displacements of lines between sets to reduce conflict misses do not suffice. This Thesis analyzes the reasons for the suboptimal behavior of the simultaneous application of DSBC and DIP [53] and proposes in a reasoned way an integrated design of these policies that allows them to cooperate effectively: the *Bimodal Set Balancing Cache (BSBC)* [61]. This extended design deals with conflict and capacity misses simul-

taneously by using the Set Saturation Level (SSL) as the unique metric to control both behaviors. Therefore, this Thesis also demonstrates the usefulness of the SSL metric to detect problems of capacity in the cache as well as problems of unbalance among the working sets of different sets. This way, the BSBC means only a 0.6% of additional storage related to the baseline cache. Simulations using benchmarks with varying characteristics show that, when properly integrated with our BSBC design, the joint application of the DSBC and BIP policies goes from being often one of the worst approaches to being the best one. For example in a 2MB, 8-way second level cache DIP+DSBC jointly reduces the miss rate by 8.3% in relative terms, while DSBC and DIP reduce it by 12% and 10% respectively. With BSBC the relative miss rate reduction almost doubles to 16%. This leads also the BSBC to get the largest IPC improvement, 4.8% on average for this configuration, compared to the 3% that a straight DSBC+DIP implementation provides. The other policies tested, DSBC, DIP and probabilistic escape LIFO lay in between. Hence, this dissertation shows that the coordination of different policies aimed to deal with different kinds of inefficiencies can be managed using cost-effective metrics and provides much larger gains that the independent application of such policies. Additionally, we have also shown that BSBC is directly applicable to shared caches, where it has performed well, achieving 10% of miss rate reduction on average and 3% of throughput improvement. This is even better than some techniques specifically designed to work on this kind of platforms, like PIPP [84]. The DSBC has been also evaluated in shared caches, reducing the baseline miss rate by 7.8% on average and achieving throughput improvements of up to 10%. Despite these positive results, techniques that include support for thread-awareness, like TADIP [27], have performed better in our experiments considering a shared LLC when the number of cores increases. This suggests the importance of giving particular treatments to different access streams in the cache.

Moreover, this Thesis shows that the idea of treating different access streams according to their individual behavior can be successfully applied to first-level caches by dynamically adjusting cache resources devoted to instructions and data, the two main streams that appear at these levels, depending on their particular demand. We propose the *Virtually Split Cache* (VSC), the first approach which is aware of the different locality instructions and data have and that allocates resources for both depending on their demand at a bank level. We have proposed two alternative

designs to track the different requirements instructions and data demand. The first approach, the *Shadow Tag* VSC, uses shadow tags to decide whether assigning one more bank for instructions or data increases performance. The second approach, the *Global Selector* VSC, uses a common saturation counter to make instructions and data fight a duel for resources. *Shadow Tag* VSC achieved 3.7% IPC improvement and 13% and 10% of miss rate and power consumption reduction related to a split baseline, respectively. *Global Selector* VSC got 3.2% IPC improvement and 11% miss rate reduction needing only 4 bits of additional storage, while achieving 8% of power consumption reduction as well. Furthermore, both VSC approaches proved to work well in multicore environments. The Shadow Tag VSC and the Global Selector VSC outperformed a 4-core baseline configuration in terms of throughput by 4.5% and 3.7% on average, respectively.

Further, this dissertation focuses on shared LLCs, where the behaviors analyzed in the preceding chapters, namely unbalances among cache sets, thrashing and the existence of different access streams, can also be found. The *Thread-Aware Bimodal Set Balancing Cache* (TABSBC), which measures the degree of pressure that each application applies to each set in the cache using the Set Saturation Level, was introduced in a sensible way. This design tackles thrashing in shared caches by including a new insertion policy, called *BIP-C*. It implies a notable improvement with respect to the original BIP [53] in these caches, as it protects lines from early displacements due to other threads. When TABSBC estimates an application is experiencing a bad cache behavior, it first tries to displace lines of the problematic application from oversubscribed sets to underutilized sets applying the Set Balancing Cache techniques. When this fails or it is not possible, it resorts to the BIP-C insertion policy for the problematic application, which has proven to be suitable to reduce capacity misses. TABSBC provides its underlying mechanisms with thread-awareness support in a coordinated and sensible way. Despite its fine-grained nature both in terms of measurement and modification of the cache behavior, and the variety of policies that it can apply, TABSBC cost is very reasonable; around 1% or less in representative configurations. Large simulations using a wide range of workloads have indicated that TABSBC consistently achieves the best overall results in comparison with recent techniques. This is due to two key characteristics that distinguish it clearly from all the other thread-aware techniques we are aware of. The first one is the small granularity at which it can take and apply decisions, as opposed

to the global decisions of other approaches. The second one is its coordinated approach to reduce conflict and capacity misses that can balance load among sets. This latter issue is largely ignored by the other proposals specifically designed for shared caches, which only focus on the workload inside each set. We think that it is of the upmost importance to identify the dead blocks in the cache and make the best usage of them. This requires changes to the placement policy like the ones explored here. For the same reasons, TABSBC scales very well as the number of cores that share the cache increases.

Finally, this dissertation reasserts the fact that a common way to increase performance in CMPs is to provide private LLCs with shared capacity by spilling lines between them. We propose *Adaptive Set-Granular Cooperative Caching* (ASCC) [62], which is able to track the state and apply different policies to each set in a cache. This is a much finer granularity than previous proposals, which apply the same policy to the whole cache. It performs spillings and applies a suitable insertion policy, if spillings are not enough to fight capacity problems, relying on the SSL of each cache set. A new insertion policy, *SABIP*, specifically designed to tackle capacity problems and be aware of displacements in environments where spillings are performed, was proposed. Also, a *neutral* state for individual sets within a spilling environment was introduced. The benefits of operating portions of the cache neither as spillers nor as receivers of lines, that is, not participating in the spilling mechanism, was successfully demonstrated. ASCC achieved 6.4% and 5.7% performance improvement running 2 and 4 cores, respectively, which translated into 18% and 21% average memory latency reductions. Its storage overhead was estimated at 0.17% related to the baseline. Furthermore, this dissertation proposes the idea of dynamically adjusting the granularity to which different policies are applied in the cache depending on its behavior. *Adaptive Variable-Granularity Cooperative Caching* (AVGCC) is the first approach able to adapt its granularity depending on the behavior of the cache to apply the ASCC policies. In a 4-core system running multiprogrammed workloads, *AVGCC* achieved a performance improvement of 7.8% with respect to the baseline. Also, it clearly outperformed recent proposals both in terms of speedup and fairness, while having a very small storage overhead, less than 0.2%. The 27% of average memory latency reduction and 29% of power consumption reduction it provided in these tests are also remarkable. Similar results were obtained in experiments with multithreaded applications. Finally, this dissertation shows that providing a given

design with *Quality of Service* (QoS) support does not always imply obtaining an average performance degradation. An extended AVGCC design, called *QoS-Aware* AVGCC, outperformed the baseline system by 8.1% despite meaning only a 0.35% of storage overhead.

7.2. Future Work

The ideas proposed in this dissertation can also be combined with other existing techniques or used for other cache related optimizations, such as reducing power consumption by deactivating underutilized cache blocks.

In this Thesis we have explored the feasibility of using information at the set level to adopt decisions on cache management. Future directions for research include tuning the size and limits of saturation counters as well as exploring other metrics to obtain a more accurate picture of the state of the cache, as other related techniques [85] do.

Although we have applied fine-grained metrics to choose the insertion policy for sets, using a global ε to guide the operation of the BIP insertion policy and its variations may not be optimal. Exploring the possibility of applying local values to the ε parameter for the proposed policies, in the different regions depending on their state, can increase performance as the application of policies at a finer structure level has successfully demonstrated in this dissertation.

Studying the different behavior in terms of performance at a set level in the VSC, or even finer granularities like line level, as well as exploring other metrics and mechanisms to track instructions and data requirements is an open line of research. Also we are planning to extend the main idea underlying the VSC design, that is, controlling the allocation of resources in the first-level cache to particular streams independently, in SMT processors at a thread level.

As future work in shared LLCs, the cooperative implementation of other policies to reduce misses can be explored. Reductions of the hardware required by our TABSBC can also be studied, for example by grouping the observation and management of nearby sets by applying the concept of variable granularity introduced

in Chapter 6.

Furthermore, the study of ASCC behavior in clusters of cores to keep a feasible degree of scalability, and the shaping of these clusters by matching complementary SSL values in different caches, may be tackled.

Appendix A

Appendix

A.1. SBC additional experiments

A.1.1. Master-Slave SBC

We have implemented an extended version of the DSBC, called Master-Slave SBC, where destination sets that become highly saturated can be recursively associated without breaking its original association (unless the break condition is fulfilled). As a result, a given set can play the role of source and destination within different associations at the same time. To achieve this, every single set has two entries in the Association Table (see Section 2.4.1), one devoted to each possible role within an association, which indicate its associated sets. Note that when a certain set plays the role of source set it can only displace native lines. Results for the two-level baseline configuration are shown in Table A.1, while results for the three-level configuration are depicted in Table A.2. Simulations were performed under the same conditions as in Chapter 2.

The improvement obtained in comparison with the original DSBC (3.5% and 5.25% average IPC improvement related to the two-level and three-level configurations, respectively) is almost negligible.

Table A.1: IPC improvement of the Master-Slave SBC over the two-level baseline configuration.

bzip2	milc	namd	gobmk	soplex	hmmer	sjeng	libquantum	omnetpp	astar	geomean
4.4%	4.2%	0.1%	0.1%	3.4%	3.6%	0.1%	2.2%	17%	4%	**3.8%**

Table A.2: IPC improvement of the Master-Slave SBC over the three-level baseline configuration.

bzip2	milc	namd	gobmk	soplex	hmmer	sjeng	libquantum	omnetpp	astar	geomean
4.4%	4.5%	3.5%	4.4%	5.4%	4.4%	4.1%	4%	16%	4.9%	**5.5%**

A.1.2. DSBC with Extra Tags

We have experimented with another extended design of the DSBC where each set has two extra tags in the Association Table (AT). These tags identify lines that have been displaced from the set to another one. The value of two tags was chosen according to the average number of displacements observed in the previous experiments (2.15, see Section 2.8.3), so that this number of tags suffices for many associations while not increasing too much the cost of the design.. It also needs a counter, which indicates the number of displaced lines to the destination set. The operation of this extended design is the following:

- When a new association is committed, the tag of the line which is displaced to the destination set is replicated in one of the two extra tags of the AT (in the entry corresponding to the source set), the other one remaining invalid. also, the counter of displaced lines is set to 1.
- A subsequent displacement would update the other extra tag and increase the counter.
- If a new displacement happens and there is no extra tag free, the counter is increased.
- Every time a cache set is accessed, all tags, including the two extra in the AT, are checked. If the counter is equal to or greater than 3 or the requested line is found in the extra tags a second search is needed.

- If a displaced line in a destination set is evicted, the counter of the source set is decreased, and one of its tags is invalidated if it corresponded to the evicted line.

As we can see the purpose of this design extension is to avoid secondary accesses under a miss in the source set of a destination. Tables A.3 and A.4 show the IPC improvement in the two-level and three-level configurations, respectively.

The percentage of improvement is slightly better than that of the Master-Slave approach but still quite low in comparison with the original DSBC.

A.2. TABSBC additional experiments

A.2.1. TABSBC using the RRIP replacement policy

The Re-Reference Interval Prediction (RRIP) [28] technique achieves good performance benefits by modifying the traditional cache replacement policy. Victim lines are selected depending on their recent behavior using a 2-bit counter to indicate the degree of reuse of each line. New lines are inserted with a reuse value of 2 or 3 depending on which option is performing better in the cache according to the set dueling mechanism. A line is selected for eviction only if its counter has a value of 3. If no such line exists, the counters for all lines in the current set are increased until one counter reaches that value. When a block is touched, its counter is set to zero (applying the *Hit Priority* approach, which is the one used in our experiments as it achieved the best results in [28]). Although a technique with thread-aware support has been proposed (TA-DRRIP, which uses set dueling to dynamically determine which option the application should apply in the presence of other applications), its efficiency is reduced when the number of applications sharing a LLC increases since a given core may evict a recently inserted line owned by a different core.

We have extended the original TABSBC design by applying RRIP as the replacement policy instead of the traditional LRU one. This way, the behavior of this version of TABSBC is listed next:

- Displaced lines must have a degree of reuse equal to 3.

Table A.3: IPC improvement of DSBC with Extra Tags over the two-level baseline configuration.

bzip2	milc	namd	gobmk	soplex	hmmer	sjeng	libquantum	omnetpp	astar	geomean
4.1%	5%	0.2%	0.2%	3.8%	4%	0.5%	2.3%	15.5%	5.2%	**4%**

Table A.4: IPC improvement of DSBC with Extra Tags over the three-level baseline configuration.

bzip2	milc	namd	gobmk	soplex	hmmer	sjeng	libquantum	omnetpp	astar	geomean
4.5%	4.5%	4%	5%	4.4%	5%	4%	4.5%	14.5%	5.2%	**5.6%**

- New lines are inserted with a degree of reuse equal to 2, if the set is applying the traditional MRU insertion, or 3, if the set is dealing with capacity problems.

Tables A.5 and A.6 show the performance improvement, measured in terms of throughput, and miss rate reduction of this version running 2 cores, respectively. The same study is performed in Tables A.7 and A.8 running 4 cores. Experiments were performed under the same conditions as in Chapter 5.

This design achieves a slight improvement related to TABSBC, which had obtained 4% IPC improvement and 12% miss rate reduction running two cores.

As for the 4-core experiments, the improvement obtained related to TABSBC, 5% performance improvement and 15% miss rate reduction, is a little bit higher than in the two-core ones.

Table A.5: Performance improvement of TABSBC with RRIP running two cores.

MW1	**MW2**	**MW3**	**MW4**	**MW5**	**MW6**	**MW7**	**MW8**
4.7%	7.9%	9.6%	6.4%	1.3%	5.2%	2.8%	4.7%
MW9	**MW10**	**MW11**	**MW12**	**MW13**	**MW14**	**MW15**	**MW16**
1%	6.3%	1.6%	6.2%	1%	1.2%	4%	1.5%
geomean				**4.1%**			

Table A.6: Miss rate reduction of TABSBC with RRIP running two cores.

MW1	**MW2**	**MW3**	**MW4**	**MW5**	**MW6**	**MW7**	**MW8**
24%	14%	17%	7%	5.1%	5%	31%	24%
MW9	**MW10**	**MW11**	**MW12**	**MW13**	**MW14**	**MW15**	**MW16**
1%	17%	2.6%	24%	5.8%	5.4%	8%	12.3%
geomean				**12.4%**			

Table A.7: Performance improvement of TABSBC with RRIP running four cores.

401+444+ 445+456	401+445+ 456+471	401+433+ 450+462	433+471+ 473+482	401+444+ 458+471	444+458+ 462+471	geomean
1.4%	3%	7.5%	14%	3%	5.4%	**5.6%**

Table A.8: Miss rate reduction of TABSBC with RRIP running four cores.

401+444+ 445+456	401+445+ 456+471	401+433+ 450+462	433+471+ 473+482	401+444+ 458+471	444+458+ 462+471	geomean
7%	20%	21%	22%	17%	19%	**17.5%**

Bibliography

[1] Model number methodology for the AMD Opteron 4100 and 6100 series processors, 2010. pages 128

[2] A. Agarwal and S. D. Pudar. Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. In *Proc. 20th Annual Intl. Symp. on Computer Architecture*, pages 179–190, May 1993. pages 5, 6, 18, 24, 72, 80

[3] A. Basu, N. Kirman, M. Kirman, M. Chaudhuri, and J. F. Martínez. Scavenger: A new last level cache architecture with global block priority. In *40th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, pages 421–432, December 2007. pages 5, 6, 19

[4] B. M. Beckmann, M. R. Marty, and D. A. Wood. ASR: Adaptive selective replication for CMP caches. In *MICRO*, pages 443–454, 2006. pages 10, 117

[5] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, pages 78–101, 1966. pages 4

[6] B. Calder, D. Grunwald, and J. S. Emer. Predictive sequential associative cache. In *Proc. of the Second Intl. Symp. on High-Performance Computer Architecture*, pages 244–253, February 1996. pages 5, 6, 18, 24, 72, 80

[7] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. In *ISCA*, pages 264–276, 2006. pages 9, 13, 117

[8] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ICS*, pages 242–252, 2007. pages 8, 10, 90, 117

[9] M. Chaudhuri. Pseudo-LIFO: The foundation of a new family of replacement policies for last-level caches. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 401–412, 2009. pages 7, 9, 48, 59, 66, 89, 91, 103

[10] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *Proc. of the 36th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, pages 55–66, December 2003. pages 6, 31

[11] Digital Equipment Corporation. Digital semiconductor 21164 alpha microprocessor product brief, March 1997. pages 25, 31

[12] H. Dybdahl and P. Stenström. An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors. In *HPCA*, pages 2–12, 2007. pages 8

[13] H. Dybdahl, P. Stenström, and L. Natvig. A cache-partitioning aware replacement policy for chip multiprocessors. In *HiPC*, pages 22–34, 2006. pages 73, 89

[14] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. In *MICRO*, pages 102–110, 1992. pages 5

[15] R. Golla. Niagara 2: A highly threaded server-on-a-chip, 2006. pages 128

[16] A. González, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *International Conference on Supercomputing*, pages 338–347, 1995. pages 5, 8

[17] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multi-processors. In *MICRO*, pages 343–355, 2007. pages 8, 90

[18] E. G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *Proc. 27th annual Intl. Symp. on Computer architecture*, pages 107–116, June 2000. pages 6, 31

[19] E. Herrero, J. González, and R. Canal. Elastic cooperative caching: an autonomous dynamically adaptive memory hierarchy for chip multiprocessors. In *ISCA*, pages 419–428, 2010. pages 9, 117, 121, 129

[20] M. D. Hill. Aspects of cache memory and instruction buffer performance. PhD thesis, 1987. pages 2

[21] HP Labs. CACTI 6.5. http://www.hpl.hp.com/research/cacti/. pages 31, 37, 60, 79, 80, 82

[22] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. *SIGARCH Comput. Archit. News*, 30(2):209–220, 2002. pages 5

[23] M. C. Huang, J. Renau, S.-M. Yoo, and J. Torrellas. L1 data cache decomposition for energy efficiency. In *ISLPED*, pages 10–15, 2001. pages 73

[24] Intel Corporation. Intel core i7 processor extreme edition and intel core i7 processor datasheet, 2008. pages 31

[25] R. R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. R. Hsu, and S. K. Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. In *SIGMETRICS*, pages 25–36, 2007. pages 8, 90

[26] A. Jaleel. Memory characterization of workloads using instrumentation-driven simulation. Retrieved on December 18, 2008, from `http://www.glue.umd.edu/~ajaleel/workload/`. pages 11, 17

[27] A. Jaleel, W. Hasenplaugh, M. K. Qureshi, J. Sebot, S. C. S. Jr., and J. S. Emer. Adaptive insertion policies for managing shared caches. In *PACT*, pages 208–219, 2008. pages xi, 9, 66, 70, 89, 91, 93, 100, 103, 147

[28] A. Jaleel, K. B. Theobald, S. C. S. Jr., and J. S. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *ISCA*, pages 60–71, 2010. pages 7, 9, 48, 58, 91, 103, 155

[29] J. Jeong and M. Dubois. Cost-sensitive cache replacement algorithms. In *HPCA*, pages 327–337, 2003. pages 5, 7

[30] T. L. Johnson, D. A. Connors, M. C. Merten, and W. mei W. Hwu. Run-time cache bypassing. *IEEE Trans. Computers*, 48(12):1338–1354, 1999. pages 8

[31] T. L. Johnson and W. mei W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *ISCA*, pages 315–326, 1997. pages 8

[32] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *ISCA*, pages 252–263, 1997. pages 5

[33] N. Jouppi and N. Wilton. CACTI: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, May 1996. pages 75

[34] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache prefetch buffers. In *Proc. 17th Intl. Symp. on Computer Architecture*, pages 364–373, June 1990. pages 5, 42

[35] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In *ISCA*, pages 240–251, 2001. pages 8

[36] S. M. Khan, D. A. Jiménez, D. Burger, and B. Falsafi. Using dead blocks as a virtual victim cache. In *Proc. 19th Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 489–500, 2010. pages 89, 104

[37] S. M. Khan, Y. Tian, and D. A. Jiménez. Sampling dead block prediction for last-level caches. In *MICRO*, pages 175–186, 2010. pages 89, 91

[38] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee. Using prime numbers for cache indexing to eliminate conflict misses. In *Proc. 10th Intl. Symp. on High Performance Computer Architecture*, pages 288–299, February 2004. pages 5, 18

[39] M. Kharbutli and Y. Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Trans. Computers*, 57(4):433–447, 2008. pages 8

[40] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *ASPLOS*, pages 211–222, 2002. pages 3

[41] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA*, pages 81–88, 1981. pages 5, 79

[42] R. Kumar and D. M. Tullsen. Compiling for instruction cache performance on a multithreaded architecture. In *MICRO*, pages 419–429, 2002. pages 73

[43] A. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *ISCA*, pages 144–154, 2001. pages 72

[44] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *ISCA*, pages 144–154, 2001. pages 5

[45] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C.-S. Kim. On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies. In *SIGMETRICS*, pages 134–143, 1999. pages 7

[46] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *MICRO*, pages 222–233, 2008. pages 72

[47] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D. yuan Chen. The performance of runtime data cache prefetching in a dynamic optimization system. In *MICRO*, pages 180–190, 2003. pages 73

[48] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *ISPASS*, pages 164–171, 2001. pages 66, 102

[49] S. McFarling. Combining branch predictors, 1993. pages 31

[50] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *HPCA*, pages 96–105, 2004. pages 5

[51] J. Peir, Y. Lee, and W. W. Hsu. Capturing dynamic memory reference behavior with adaptive cache topology. In *Proc. of the 8th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 240–250, October 1998. pages 6, 20

[52] M. K. Qureshi. Adaptive spill-receive for robust high-performance caching in CMPs. In *HPCA*, pages 45–54, 2009. pages 9, 13, 117, 121, 123

[53] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely Jr., and J. S. Emer. Adaptive insertion policies for high performance caching. In *Proc. 34th Intl. Symp. on Computer Architecture*, pages 381–391, June 2007. pages xi, xii, 7, 9, 12, 35, 36, 47, 48, 49, 50, 51, 69, 73, 91, 92, 100, 113, 122, 129, 146, 148

[54] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, pages 423–432, 2006. pages 8, 73, 89, 90, 91, 92

[55] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-Way cache: Demand-based associativity via global replacement. In *Proc. 32st Intl. Symp. on Computer Architecture*, pages 544–555, June 2005. pages 6, 11, 17, 19, 20, 31, 117, 119

[56] P. Racunas and Y. N. Patt. Partitioned first-level cache design for clustered microarchitectures. In *ICS*, pages 22–31, 2003. pages 73

[57] M. V. Ramakrishna, E. Fu, and E. Bahcekapili. Efficient hardware hashing functions for high performance computers. *IEEE Transactions on Computers*, 46(12):1378–1381, 1997. pages 84

[58] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net. pages 30, 79, 102, 128

[59] J. A. Rivers, G. S. Tyson, E. S. Davidson, and T. M. Austin. On high-bandwidth data cache design for multi-issue processors. In *MICRO*, pages 46–56, 1997. pages 77

[60] D. Rolán, B. B. Fraguela, and R. Doallo. Adaptive line placement with the Set Balancing Cache. In *Proc. 42nd IEEE/ACM Intl. Symp. on Microarchitecture*, pages 529–540, December 2009. pages x, 49, 69, 73, 117, 119, 146

[61] D. Rolán, B. B. Fraguela, and R. Doallo. Reducing capacity and conflict misses using Set Saturation Levels. In *Proc. 17th Intl. Conf. on High Performance Computing*, December 2010. pages xi, 73, 117, 119, 146

[62] D. Rolán, B. B. Fraguela, and R. Doallo. Adaptive set-granular cooperative caching. In *HPCA*, pages 213–224, 2012. pages xiii, 149

[63] A. Samih, Y. Solihin, and A. Krishna. Evaluating placement policies for managing capacity sharing in cmp architectures with private caches. *TACO*, 8(3):15, 2011. pages 9

[64] D. Sanchez and C. Kozyrakis. The zcache: Decoupling ways and associativity. In *MICRO*, pages 187–198, 2010. pages 6, 18

[65] A. Seznec. A case for two-way skewed-associative caches. In *Proc. 20th Annual Intl. Symp. on Computer Architecture*, pages 169–178, May 1993. pages 5, 7, 18

[66] A. J. Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982. pages 2

[67] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *ASPLOS*, pages 234–244, 2000. pages 66, 102

[68] K. So and R. N. Rechtschaffen. Cache operations by MRU change. *IEEE Trans. on Computers*, C-37(6), June 1988. pages 2

[69] G. S. Sohi and M. Franklin. High-bandwidth data memory systems for superscalar processors. In *ASPLOS*, pages 53–62, 1991. pages 75, 77

[70] S. Srikantaiah, M. T. Kandemir, and M. J. Irwin. Adaptive set pinning: managing shared caches in chip multiprocessors. In *ASPLOS*, pages 135–144, 2008. pages 8, 90, 91

[71] S. Srikantaiah, E. Kultursay, T. Zhang, M. Kandemir, M. Irwin, and Y. Xie. Morphcache: A reconfigurable adaptive multi-level cache hierarchy. In *HPCA*, pages 231–242, 2011. pages 10, 117

[72] S. T. Srinivasan, R. D.-C. Ju, A. R. Lebeck, and C. Wilkerson. Locality vs. criticality. In *ISCA*, pages 132–143, 2001. pages 7

[73] C.-L. Su and A. M. Despain. Cache design trade-offs for power and performance optimization: a case study. In *ISLPD*, pages 63–68, 1995. pages 75, 77

[74] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004. pages 89

[75] J. M. Tendler, J. S. Dodson, J. S. Fields, Jr., H. Le, and B. Sinharoy. Power4 system architecture. In *IBM Journal of Research and Development*, 2007. pages 80

[76] J. Torrellas, C. Xia, and R. L. Daigle. Optimizing the instruction cache performance of the operating system. *IEEE Trans. Computers*, 47(12):1363–1381, 1998. pages 73

[77] G. S. Tyson, M. K. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *MICRO*, pages 93–103, 1995. pages 8

[78] T. Wada, S. Rajan, and S. Przybylski. An analytical access time model for on-chip cache memories. *IEEE Journal of Solid-State Circuits*, 27(8):1147–1156, August 1992. pages 75

[79] D. Weiss, J. Wuu, and V. Chin. The on-chip 3-MB subarray-based third-level cache on an itanium microprocessor. *IEEE journal of Solid State Circuits*, 37(11):1523–1529, November 2002. pages 25, 31

[80] M. Wilkes. Slave memories and dynamic storage allocation. *IEEE Transactions on Computers*, EC-14:270–271, 1965. pages 2

[81] W. A. Wong and J.-L. Baer. Modified lru policies for improving second-level cache behavior. In *HPCA*, pages 49–60, 2000. pages 8

[82] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. S. Jr., and J. S. Emer. Ship: signature-based hit predictor for high performance caching. In *MICRO*, pages 430–441, 2011. pages 89

[83] C.-J. Wu and M. Martonosi. Characterization and dynamic mitigation of intra-application cache interference. In *ISPASS*, pages 2–11, 2011. pages 112

[84] Y. Xie and G. H. Loh. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *ISCA*, pages 174–183, 2009. pages xi, 9, 66, 70, 89, 91, 103, 147

[85] D. Zhan, H. Jiang, and S. C. Seth. STEM: Spatiotemporal management of capacity for intra-core last level caches. In *MICRO*, pages 163–174, 2010. pages 7, 73, 117, 119, 150

[86] C. Zhang. Balanced cache: Reducing conflict misses of direct-mapped caches. In *Proc. 33rd Intl. Symp. on Computer Architecture*, pages 155–166, June 2006. pages 6, 72

[87] C. Zhang, X. Zhang, and Y. Yan. Two fast and high-associativity cache schemes. *IEEE MICRO*, 17:40–49, 1997. pages 6, 18, 72, 80

[88] M. Zhang and K. Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *ISCA*, pages 336–345, 2005. pages 8