UNIVERSIDAD DE MURCIA

FACULTAD DE INFORMÁTICA

Improving the Energy-Efficiency of
Cache-Coherent Multi-Cores.

Técnicas para Mejorar la Eficiencia Energética de
los CMPs con Coherencia de Caché.

D. Antonio García Guirado
2013

Universidad de Murcia
Departamento de Ingeniería y
Tecnología de Computadores

Improving the Energy-Efficiency of Cache-Coherent Multi-Cores

A dissertation submitted in fulfillment of
the requirements for the degree of

Doctor en Informática

By
Antonio García Guirado

Advised by
José Manuel García Carrasco
Ricardo Fernández Pascual

Murcia, September 2013

Abstract

With the end of the energy benefits of classical transistor scaling, energy-efficiency became the driving factor of microprocessor design. The energy costs of increasing operating frequencies and using more aggressive microarchitectures to improve performance can no longer be compensated by transistor technology advances at the same rate as in the past. Now more than ever, computer architects are left in charge of providing the expected growth of microprocessor performance by means of more energy-efficient designs, in order to make cost-effective use of the ever-increasing transistor density in chips within a limited power budget. Otherwise, microprocessor evolution would be doomed to hit a *power wall*.

Some years back, chip multiprocessors (CMPs) containing relatively simple cores were deemed a suitable architecture in the search for energy-efficiency and went mainstream. However, CMPs pose particular challenges in their design. In this thesis we tackle two paramount issues with growing importance as we scale out CMPs, namely network-on-chip power consumption and cache coherence scalability, and we propose mechanisms to alleviate their impact on the energy-efficiency of CMPs.

Cache coherence information may eventually account for most storage area on chip if we fail to provide scalable and energy-efficient coherence schemes. To address this challenge, we propose a new coherence scheme based on a chip divided in linked areas that noticeably reduces the size of the storage used to maintain sharing information. Then, we propose a unified cache organization that eliminates the overhead and complexity of directory storage structures by leveraging shared cache resources to alternately store data or directory information at a cache entry granularity. This organization makes efficient use of storage resources as just the required (small) number of entries are allocated for directory information in the unified cache.

The second CMP design challenge tackled by this thesis is that, as the core count increases, current trends indicate that networks-on-chip (NoCs) would end up taking up most of the chip energy budget unless measures are taken to prevent it. These measures include the development of techniques to increase the proximity of data to cores at the architecture level and the use of disruptive low-power transmission technologies such as silicon photonics. For increasing data proximity, in this thesis we propose a mechanism that retrieves data from a close provider in the area in which the core requesting the data is located, working at the cache coherence protocol level. At the cache organization level, we propose a new organization that aims at minimizing the average distance to access the last level cache, reducing the energy required to feed the cores with data. To enable the exploitation of photonics, we open the field of dynamic policies for arbitrating hybrid photonic-electronic NoCs, making energy-efficient use of the particular properties of both technologies at a fine message granularity, resulting in important benefits in throughput, energy and latency.

We expect that the mechanisms described in this thesis will help scale-out CMPs in the current scenario of energy-constrained transistor miniaturization, in order to achieve the ever-increasing performance demanded of computing systems.

Acknowledgments

I would like to start by thanking my advisors, Ricardo and Jose Manuel. First they gave me the chance to start researching in the field I liked since I started my undergraduate studies, and then they were by my side throughout all these years, making this such an enriching experience. As a preparatory stage to performing research, I dare say that they made my doctorate years successful.

I also want to thank all my laboratory mates: Rubén, Dani, Ginés, Chema, Juanma, as well as those in the laboratory next door: Dani, Epi, José Luis, Antonio, Ana, and other colleagues: Alberto, Varadan, for sharing these years with me and helping me in everything I needed. I want to express my gratitude to my host at the Università degli Studi di Siena, Sandro Bartolini, for making me feel at home for almost five months and supporting me ever since. Also, thanks to Pierfrancesco Foglia and Sylvain Collange for taking on the extensive task of reviewing my thesis. I must remark and thank the support I received from my friends in Murcia and as well as in Siena; they are too many to be listed here. Still, Ángel, Cai, Fernando, José David, Josipa, Rafah and Stefania have to appear by name. Last, but most important, I thank my family for everything.

Contents

Índice

List of Figures

List of Tables

Resumen

Introducción y Motivación

En las últimas décadas, la computación moderna ha proporcionado constantes progresos en todas las áreas de nuestras vidas. Esto ha sido posible gracias a un continuo crecimiento exponencial del rendimiento de los microprocesadores. Desafortunadamente, la informática se enfrenta a un enorme reto a día de hoy debido a limitaciones físicas de la tecnología actual que suponen una barrera al incremento de rendimiento al que nos hemos acostumbrado.

Durante la mayor parte de la evolución de los microprocesadores desde que el primero fuera fabricado en 1971 (el Intel 4004), su rendimiento ha mejorado mediante el uso de frecuencias de reloj más altas y diseños de arquitectura más complejos, posibilitados por el continuo desarrollo de transistores más pequeños y más rápidos. En cada nueva generación, los procesadores eran capaces de explotar más paralelismo a nivel de instrucción (ILP) de manera dinámica y de hacerlo a una frecuencia de operación mayor, doblando su rendimiento cada dos años aproximadamente.

Este crecimiento exponencial de rendimiento fue posible gracias a que el escalado clásico de los transistores asociaba a la reducción de tamaño una reducción exponencial del consumo energético. Esto mantenía la densidad de potencia disipada en los nuevos chips dentro de unos límites manejables a pesar del aumento de frecuencia y de complejidad en la circuitería, permitiendo siempre reducir la energía disipada por operación ejecutada. Esta constante reducción de la energía por operación ha sido la piedra angular sobre la que se ha sustentado la evolución de la informática. Si la miniaturización no hubiese llevado asociada tal mejora energética en los transistores, la potencia necesaria para alimentar servidores como los de hoy en día sería inaceptable, los dispositivos móviles modernos vaciarían rápidamente sus baterías, y además ningún medio de refrigeración razonable sería capaz de evitar que estos chips ardiesen literalmente, haciendo imposible su existencia.

Desafortunadamente, aunque aún somos capaces de integrar el doble de transistores en el mismo área cada 18–24 meses, en cierto momento (alrededor de 2004, con el paso de un tamaño de transistor de 65 nm a uno de 45 nm) cuestiones físicas nos hicieron incapaces de seguir escalando las características energéticas de los transistores de la manera clásica, perdiendo gran parte de los beneficios asociados. Esta incapacidad puso y sigue poniendo en riesgo futuras reducciones de la energía consumida por operación, y por tanto futuros avances en el rendimiento de los procesadores.

Para continuar mejorando el rendimiento con estas restricciones energéticas no podemos seguir confiando en la miniaturización de los transistores como antes, pues ésta ya no puede contrarrestar los costes energéticos de incrementar la frecuencia y usar simultáneamente más transistores en diseños más complejos. Por tanto, ahora es el propio diseño de los chips el que debe tener como objetivo la eficiencia energética, para así seguir reduciendo la energía consumida por operación y poder proporcionar más rendimiento sin elevar el consumo energético. Es por ello que los multiprocesadores en un chip (CMPs), que integran varios cores relativamente simples, fueron adoptados como una alternativa más eficiente energéticamente que los procesadores que integran un único núcleo más complejo. La idea detrás de este cambio radica en que aumentar la complejidad del diseño de un core incrementa su rendimiento de manera proporcional a la raíz cuadrada del incremento en el número de transistores empleados, y a la vez la mayor complejidad de los circuitos resultantes, necesaria para poder identificar y ejecutar un mayor número de operaciones en paralelo, aumenta el factor de actividad de los transistores (es decir, el número de transistores que se activan simultáneamente para poder realizar cada operación). Mayor número de transistores y mayor factor de actividad significan un crecimiento insostenible del consumo energético por operación. En comparación, replicar el diseño del mismo core colocando varios cores adyacentes puede aumentar el rendimiento de manera proporcional al número de transistores empleados y sin aumentar el factor de actividad de los transistores. Si queremos hacer realidad este ideal de crecimiento de rendimiento proporcional al área y al consumo energético, lo que permitiría mejorar el rendimiento con avances en la tecnología de fabricación más modestos, debemos superar los retos que plantea la existencia de un número cada vez mayor de cores en el chip.

En particular, explotar el paralelismo a nivel de hilo (TLP) proporcionado por números de cores cada vez mayores requiere aplicaciones paralelas que son difíciles de programar. No es ninguna sorpresa que el modelo de programación paralela más extendido sea a la vez el más simple: memoria compartida. Los modelos de memoria compartida son normalmente implementados sobre un protocolo de coherencia de caché, que es necesario para realizar el manejo implícito de las memorias caché privadas que ocultan la mayor parte de la diferencia de velocidad entre los cores y la lenta memoria DRAM principal. Esto crea la necesidad de luchar contra los problemas de escalabilidad de los mecanismos de coherencia de caché para no crear otra barrera a la eficiencia energética. En particular, el porcentaje de área ocupada por los mecanismos de coherencia considerados más escalables (basados en directorio) crece proporcionalmente al número de cores.

Un segundo problema particular de los CMPs es que los multiprocesadores son especialmente propensos a movimientos de datos entre distintas partes del chip, lo que a su vez puede reducir la eficiencia energética, llegando a consumir la mayor parte de la energía en transmisión de datos. Si no se explota adecuadamente la localidad de datos, y si no se adoptan nuevas tecnologías de transmisión más eficientes, el consumo energético de la comunicación on-chip puede convertirse en un serio problema para la evolución de la informática.

En esta tesis hemos abordado estos dos retos específicos de los multicores, la escalabilidad del protocolo de coherencia y el consumo energético de la red de interconexión. Como modelo de estudio hemos usado un diseño tiled-CMP en el que el rendimiento aumenta mediante replicación de *tiles*, bloques básicos que contienen un core, recursos de caché y una interfaz de red para comunicarse mediante una red escalable como una malla. En particular, las siguientes oportunidades motivan esta tesis:

- El protocolo de coherencia de caché coordina la comunicación en el chip y puede optimizarse para mejorar la proximidad de los datos, también permitiendo un tamaño reducido para códigos de compartición exactos para mejorar la escalabilidad del mecanismo de coherencia.

- Las organizaciones de caché se pueden optimizar para mejorar la proximidad de los datos reorganizando los mapeos que determinan qué cores acceden a qué bancos de caché, reduciendo la distancia y el número de retransmisiones en la red.

- Los recursos compartidos de la caché pueden usarse más eficientemente para almacenar datos e información de compartición, mejorando la escalabilidad de la coherencia de caché.

- Las tecnologías fotónica y electrónica pueden combinarse para explotar las mejores características de cada una de ellas, aumentando la eficiencia de la red con respecto a usar tan sólo una de estas tecnologías, y se necesitan políticas de gestión adecuadas para hacerlo eficientemente.

Como resultado de la investigación realizada proponemos mecanismos para reducir el impacto de la coherencia de caché y el consumo de la red de interconexión. Las contribuciones de esta tesis se resumen en los siguientes apartados.

Protocolos de Coherencia con Multiples Areas

Proponemos una familia de protocolos de coherencia que dividen el chip en áreas estáticamente para reducir la sobrecarga de almacenamiento y mejorar la proximidad entre los datos y los cores, consiguiendo reducir el consumo energético y acelerar la operación del chip. Estos protocolos mantienen información de coherencia por área y utilizan punteros que enlazan las áreas para mantener un único dominio de coherencia en todo el chip. Este esquema de almacenamiento proporciona una reducción de espacio de almacenamiento que es especialmente significativa para grandes números de cores (e.g., 93 % de reducción comparado con un directorio de vector de bits full-map para 1024 cores y usando tan solo 4 áreas).

Además, el protocolo de coherencia selecciona dinámicamente un nodo por área para actuar como proveedor de los datos para los fallos de caché que se produzcan en su área. Los cores usan un mecanismo de predicción para alcanzar al proveedor del área para cada bloque, evitando la indirección típica del directorio y reduciendo la latencia media de los fallos de caché y el tráfico de red.

Comparado con un directorio de vector de bits full-map, estos protocolos reducen el consumo estático del directorio un 54 % y el consumo dinámico de las cachés y la red hasta un 38 % en un multiprocesador de 64 cores con cuatro áreas, sin mostrar degradación del rendimiento, y hasta mejorándolo un 6 % en el servidor web apache. Estos protocolos fueron propuestos en el contexto de la consolidación de servidores por medio de virtualización, pero son igualmente aplicables a cargas multiprogramadas y aplicaciones paralelas.

Organización de Caché Parcialmente Compartidas Basada en Distancia (DAPSCO)

Cuanto mayor sea el número de cores en un chip, más interesantes resultan los diseños parcialmente compartidos para el último nivel de caché (LLC). En una caché parcialmente compartida tradicional los cores comparten sus bancos de LLC en grupos, resultando en un diseño que causa menos accesos a memoria que una LLC totalmente privada por core y menos tráfico de red y menor latencia que una LLC totalmente compartida entre todos los cores. En una organización de caché parcialmente compartida el número de cores que forman cada grupo y comparten sus bancos de LLC es conocido como el *grado de compartición* de la caché.

En este contexto, nosotros proponemos DAPSCO como una optimización a las cachés parcialmente compartidas tradicionales, basándonos en la observación de que agrupar los bancos de LLC y cores en grupos no es la organización de caché más eficiente. Para ilustrar las ineficiencias del esquema tradicional, basta con observar que un core situado en la esquina de un grupo se encuentra de media notablemente más lejos de los bancos de LLC del grupo que un core situado en el centro. DAPSCO usa un mapeo entre cores y bancos de LLC más eficiente en el que no existen grupos y cada core accede a los bancos de LLC que le rodean, minimizando así la distancia media a la LLC.

El mapeo de DAPSCO mantiene las mismas propiedades deseables de las organizaciones tradicionales: los bancos de LLC almacenan subconjuntos del espacio de memoria y cada core accede a un banco por cada subconjunto para alcanzar el espacio de memoria completo.

Hemos generado y evaluado configuraciones de DAPSCO para números de cores desde 64 hasta 512 con distintos grados de compartición y topologías de red. Por ejemplo, DAPSCO reduce el número medio de enlaces para alcanzar la LLC un 33 % en un CMP de 512 cores con un grado de compartición de 128, reduciendo la latencia de los accesos LLC y el uso de energía de manera similar.

DAPSCO fue evaluado mediante simulación detallada en un CMP de 64 cores para confirmar sus ventajas teóricas. Además, DAPSCO casi no introduce sobrecarga, ya que tan sólo requiere cambios menores en algunos circuitos combinacionales ya presentes en el chip.

Información de Coherencia Almacenada en Caché (ICCI)

ICCI es una nueva organización de caché que hace que la sobrecarga de almacenamiento asociada a mantener un directorio sea escalable. ICCI se basa en almacenar entradas de directorio en los recursos de caché compartidos ya existentes en el chip, proporcionando coherencia de caché hardware económica para gran número de cores (e.g., 512). ICCI consigue tiempos de ejecución cercanos a un directorio no escalable y reduce a la vez el consumo de energía del sistema de memoria notablemente. La idea detrás de ICCI es que conforme aumenta el número de cores y el grado de compartición de los datos también lo hace, menos entradas de directorio por core son necesarias, ya que cada entrada mantiene la información de compartición del mismo bloque en más cachés privadas.

ICCI hace un uso más eficiente de los recursos de caché permitiendo de manera dinámica que las entradas del último nivel de caché almacenen bloques o códigos de compartición de manera alternativa, resultando en un diseño más compacto, sencillo y eficiente que si se usa una estructura dedicada para bloques y otra para información de directorio. Mediante el aprovechamiento dinámico de entradas de la LLC para almacenar información de directorio, solo el número de entradas realmente necesarias para información de compartición son reservadas en cada momento, dejando el resto de entradas libres para almacenar datos.

Hemos evaluado analíticamente que el uso de recursos de ICCI para información de directorio es asumible para grandes números de cores. Con grados de compartición encontrados en la literatura ICCI usa incluso menos espacio que alternativas escalables en términos de área (pero no de rendimiento o consumo energético) como SCI o tags duplicados. De hecho, contrariamente a otros esquemas, ICCI usa menos espacio para almacenar información de directorio conforme aumenta el número de cores, debido a la mayor compartición y al tamaño fijo de las entradas usadas (el tamaño de una entrada de la LLC).

En caso de no existir compartición de datos, el uso de mecanismos ya propuestos para mantener el estado privado de las páginas en la tabla de páginas hace que ICCI no use ningún espacio en LLC para información de directorio para las páginas privadas, lo cual es una característica única hasta el momento. Otra ventaja de ICCI es su simplicidad: cambios muy sencillos en el sistema con respecto a un directorio tradicional de vector de bits son suficientes para implementar ICCI, lo que contrasta con la complejidad de otras propuestas.

Además, el diseño de ICCI garantiza que se producirá una cantidad despreciable de invalidaciones de bloques activos causadas por reemplazos de directorio comparado con el gran volumen de éstas que pueden producirse potencialmente en los esquemas tradicionales. Los resultados para un CMP de 512 cores muestran que ICCI reduce el consumo de energía del sistema de memoria en hasta un 48 % comparado con un directorio embebido en los tags, y de un 8 % comparado con el recientemente propuesto Directorio de Coherencia Escalable (Scalable Coherence Directory, SCD), al que ICCI también mejora en tiempo de ejecución de las aplicaciones y área utilizada, resultando en un efecto multiplicativo en la mejora en eficiencia energética con respecto a SCD. Además, ICCI puede usarse en combinación con códigos de compartición más elaborados para aplicarlo a números de cores extremadamente altos (e.g., 1 millón).

Políticas Dinámicas de Gestión para Redes Híbridas Fotónicas-Electrónicas

Para mejorar el aprovechamiento de la reciente tecnología fotónica en silicio, que necesita integrarse con la tecnología electrónica sobre la que se construyen los chips, proponemos políticas dinámicas para el manejo eficiente de redes híbridas fotónicas-electrónicas. Al contrario que la electrónica, la fotónica es eficiente para comunicaciones de larga distancia en un chip, ya que la luz no requiere de retransmisiones para alcanzar su destino. Esto puede aprovecharse para mitigar los efectos de un mayor número de cores en las transmisiones, haciendo que no se requiera un número creciente de retransmisiones intermedias. Por otra parte, la electrónica sigue teniendo buenas características para comunicar elementos próximos, ya que al contrario que la fotónica no requiere de conversiones entre ambas tecnologías. Esto puede aprovecharse para evitar usar los valiosos recursos de la red fotónica en transmisiones cercanas que no obtienen beneficios. Hasta ahora, las redes híbridas utilizaban políticas estáticas para decidir si la transmisión de un mensaje se realizaría de manera electrónica o fotónica, siendo incapaces de adaptarse a las condiciones de funcionamiento del chip en cada momento, resultando en un uso de recursos ineficiente que puede degradar el rendimiento o aumentar el consumo energético innecesariamente. En esta tesis hemos desarrollado políticas que utilizan información en tiempo real para realizar la decisión de qué tecnología utilizar para la transmisión con granularidad de mensaje. Hemos probado varias de estas políticas, usando como punto inicial una malla electrónica (usada comúnmente en chips que contienen decenas de cores) combinada con una red fotónica basada en anillos (que aprovecha la velocidad de propagación de la luz para realizar transmisiones rápidas), utilizando tanto aplicaciones reales como tráfico sintético. Un modesto anillo basado en FlexiShare fue usado como un escenario cercano en el futuro, y diseños más grandes basados en Corona y Firefly se usaron como redes a largo plazo para 256 cores. Combinando información como el tamaño del mensaje, el tiempo esperado para transmisión fotónica y distancia entre elementos, hemos desarrollado una política que obtiene grandes beneficios de energía y rendimiento comparada con maneras alternativas de manejar la red híbrida. Esta política consigue una capacidad de transmisión en número de mensajes para la red híbrida hasta un 38 % superior a la suma de la de las subredes fotónica y electrónica por separado, en un escenario con 64 puntos de acceso a la red. También obtiene una latencia media más baja que cualquiera de las subredes realizando transmisiones a corta distancia usando la malla electrónica y a larga distancia usando la subred fotónica.

Conclusiones y Vías Futuras

Conforme aumenta el número de cores en los multiprocesadores también aumenta la importancia de la escalabilidad de los esquemas de coherencia de caché y de la red de interconexión. En esta tesis hemos propuesto mecanismos que reducen la sobrecarga de almacenamiento de la coherencia de caché, formas de reducir la distancia hasta los datos accedidos y políticas para aprovechar de manera más eficiente una mezcla de recursos fotónicos y electrónicos para la transmisión de datos en el chip. Esperamos que estas propuestas ayuden a mejorar la escalabilidad de los CMPs para que puedan hacer efectivos sus beneficios teóricos.

Además, el trabajo desarrollado en esta tesis abre varias vías futuras de investigación:

- Políticas de remplazo para protocolos de coherencia con múltiples áreas que tengan en cuenta la proximidad del bloque a reemplazar para potenciar la efectividad de los proveedores que más beneficio proporcionen y la exactitud de predicción de los mismos.

- Protocolos de coherencia de caché para aumentar la proximidad de datos más elaborados mediante el uso de árboles en que cada nodo almacena información de coherencia recursivamente sobre dos o más hijos. Teniendo en cuenta la distancia entre nodos al insertar cada compartidor, haciéndolo en la rama con el hijo más cercano en cada bifurcación del árbol, y utilizando predicción al nodo padre como proveedor, podemos alcanzar a un proveedor muy cercano en los fallos de caché (a menudo adyacente en el chip). El árbol además proporciona implícitamente un mecanismo de multicast en envío de invalidaciones y recolección de respuestas.

- Combinación de los protocolos de múltiples áreas y DAPSCO en un nuevo diseño con las mejores características de ambos. A la hora de formar las áreas puede utilizarse un diseño similar a DAPSCO para reducir la distancia a los proveedores.

- Estudio de formas de integrar ICCI con DAPSCO y los protocolos de múltiples áreas para obtener mayores reducciones en almacenamiento de coherencia.

- Cachés privadas-compartidas dinámicas con bajo uso de recursos para coherencia de caché mediante ICCI. ICCI puede almacenar el mínimo número necesario de entradas de directorio en la LLC, permitiendo el reemplazo selectivo de bloques de manera local sin necesidad de utilizar un enorme directorio para almacenar la información de cachés LLC privadas.

- Estudio de políticas de gestión para redes híbridas para mejorar su eficiencia al trabajar con diferentes organizaciones de caché y protocolos de coherencia, evitando así la necesidad de desarrollar topologías específicas para distintos chips.

Chapter 1

Introduction

In the last decades, modern computing technology has steadily brought immense progress to all areas of life. This progress, which will hopefully continue in the future, has been enabled by maintained exponential growth of microprocessor performance. Unfortunately, computing faces an enormous challenge today, as physical limitations of current technology pose barriers to sustaining the performance increase to which we have grown accustomed.

For most of the evolution of microprocessors since they were first introduced in 1971 [49], their performance was improved by using higher clock rates and ever more complex architecture designs, which were enabled by faster and smaller transistors regularly available [41]. In every new generation, processors were able to exploit more instruction level parallelism (ILP) dynamically [153], and do so at a higher operating frequency, approximately doubling their performance every two years.

This exponential performance growth was possible thanks to the almost-too-good-to-be-true energy improvements provided by classical transistor scaling [43] that always kept the power density of chips within manageable limits, constantly reducing the energy dissipated to execute a single operation. This sustained reduction in energy per operation has been the cornerstone of computing evolution [106]. Without such transistor energy improvements, the power required to operate today's powerful computers would be unacceptable, modern mobile devices would quickly drain their batteries, and reasonable cooling means would be unable to prevent these chips from literally catching fire in the first place.

Unfortunately, even though we are still able to integrate twice as many transistors in the same area every 18–24 months, at a certain point, physical limits just made us unable to continue achieving the same energy benefits as before [74]. This inability jeopardizes further energy per operation reductions and hence future advances in processor performance, apparently heading us toward a future similar to the situation previously described if we try to improve performance at the same pace as in the past.

To keep on improving performance with these power limitations we cannot rely as much on transistor miniaturization as before, as it can no longer compensate for the power costs of historical trends of higher operating frequencies and simultaneous use of more transistors in more complex designs. Hence, chip designs striving for higher energy-efficiency are now required to reduce energy per operation and enable better performance by other means. This was understood some years ago, and chip multiprocessors (CMPs) containing a number of simpler cores were adopted as an energy efficient alternative to complex single-core processors, posing their own design challenges when scaling them out to large numbers of cores [64].

In particular, exploiting thread level parallelism (TLP) by increasing the number of cores requires hard-to-code parallel applications. It is no surprise that the most extended parallel programming model is also the simplest one: shared memory [3]. Shared memory models are most often implemented on top of cache coherence protocols [125], which are needed to perform implicit management of the private cache memories that hide most of the speed gap between cores and slow DRAM main memory. This creates a need to fight the scalability issues of cache coherence mechanisms in order to avoid creating another barrier to energy-efficiency.

A second particular problem of CMPs is that they are especially prone to data movements between different parts of the chip, which can in turn reduce the energy efficiency of the chip by consuming most of its power budget performing communication instead of computation [27]. If data proximity is not adequately exploited or new energy-efficient transmission technologies are not adopted, the energy consumption of on-chip communication can become the limiting factor for computing evolution.

Hence, the current situation calls for architectural techniques to reduce the energy consumption of CMPs in order to avoid the bleak perspective of *dark silicon* in the future, which would result in chips endowed with huge amounts of resources of which just small parts can be turned on at a time due to power constraints [47]. Efficient architecture designs should increase the performance per area and power unit, as well as reduce power density to raise the number of transistors that can be used simultaneously, allowing computing to progress and continue to improve our lives. Now more than ever, it is the turn of computer architects to fill in for the slower energy scaling of transistors.

The following sections of this chapter give more insight into the current situation of microprocessors, motivate this thesis and summarize its contributions.

1.1 The Power Wall and the Shift to Multi-Core

A long list of innovations separates the first microprocessor, the Intel 4004 released in 1971 [48], and today's chips such as the latest Intel Haswell Core i7 family of processors, launched in 2013. While the Intel 4004 integrated just 2300 transistors with a 10000 nm feature size and worked at 740 KHz, the latest Core i7 is a multi-billion transistor chip with a 22 nm feature size integrating four cores working at up to 4 GHz, and including dedicated resources for many other functions such as graphics processing. In comparison, the Intel 4004 was able to execute up to 92000 4-bit operations per second while a modern Core i7 executes in the order of 2×10^{11} 64-bit integer operations per second. It would not be inaccurate to say that transistor miniaturization in itself accounts for most of the increase in the performance of microprocessors.

For decades, classical transistor miniaturization provided the beneficial effects expected from the application of Dennard's scaling rules [43]. With each new generation, appearing approximately every 18–24 months, transistor dimensions shrank by 30%, resulting in equivalent 50% smaller area per transistor. This brought associated benefits by applying Dennard's rules. For one, 30% smaller dimensions translated directly into a 30% transistor delay reduction, enabling the operation of the same processor designs with equivalent 30% shorter CPU cycles —or 40% faster clock rate— than with the previous generation of transistors, boosting performance accordingly. In addition, supply voltage was reduced by 30% to keep the electronic field constant —and transistor capacitance also decreased 30%—, which reduced transistor power consumption by 50% after the switching frequency increase was applied. Overall, power density was kept constant on the chip, but transistors worked 40% faster and there were twice as many of them in the same area as in the previous generation.

More transistors available meant that more complex architectures were implementable. Computer architects designed larger cores to exploit more and more instruction level parallelism (ILP) of applications, progressively incorporating architecture advances such as on-die caches [65,163], pipelining [38,104], superscalar processing [30,164], and out-of-order and speculative execution [20,169,179]. These architectural advances enabled roughly another 40% increase in performance with each doubling of the number of transistors within a reasonable power budget. However, extracting more ILP translated into increased activity factors of the transistors due to the higher complexity of the circuitry, raising the overall power consumption of the chip.

In summary, higher operating frequencies and more complex designs made it common to expect twice as powerful processors every two years. Most of such performance improvement, and especially the underlying energy per operation reduction, were exclusively brought forth by classical transistor energy scaling. It was becoming increasingly difficult to develop new microarchitectural advances for every technology generation, as extracting more ILP from sequential code was becoming more and more complex, due to the intrinsically limited ILP available in applications as well as to the growing complexity involved in the associated logic to uncover extra ILP [175].

Even benefiting from classical transistor energy scaling, the power dissipated by chips was steadily growing for a number of reasons, some of which had more to do with aggressive competition to release the fastest processor than with incapability to keep low power dissipation. In addition to the increasing activity factors of transistors caused by ever more aggressive microarchitectural optimizations, chip area was also growing to allow even larger cores and caches for extracting more ILP, and frequency was increasing over what was recommended by Dennard's scaling rule (that is, over the factor by which transistor delay and voltage were reduced) in a fierce fight to sell the *apparently* fastest processor.

One of the last techniques in the trend to create more complex cores was Intel Pentium 4's deep-pipelining (with up to 31 pipeline stages in 2004's Prescott Pentium 4) to provide ultra-high operating frequency, which was very marketable at the time, and extract more ILP [83]. The extreme power consumption of deep-pipelining exemplified the ever-increasing difficulties to exploit more transistors to extract ILP in a single thread, especially in an energy-efficient way. This path for seeking increased performance progressively brought power dissipation up from Intel 4004's modest 0.63 W to Pentium 4's staggering 130 W, despite benefiting from the energy benefits of classic transistor scaling. Nevertheless, exponential performance improvements were always attained within a reasonable power budget at the time, and fortunately, energy per operation was always going down thanks to technology advances.

When exploiting ILP proved insufficient, chip designers started to turn to thread level parallelism (TLP), with the introduction of multithreading (SMT [116]), to make more efficient use of the huge amount of available resources in the chip that were poorly used by ILP-oriented architectures, by executing several threads in the same core at the same time, in a presage of what was to come.

Around 2004, at a 90 nm feature size, the energy benefits of classical technology scaling faded, and they were no longer applicable by reducing voltage when scaling down to 65 nm [74] due to the appearance of large leakage currents. This fading coincided with the end of the lifetime of the ultra-aggressive power-hungry NetBurst design of Intel's Pentium 4, exacerbating the situation. Feature sizes could still be shrunk at the same rate, which steadily brought us to the 22 nm era today [8, 25], with 14 nm fabrication plants currently under development. No theoretical scaling limit has been seen yet for silicon, and projections expect further miniaturization down to 5 nm for CMOS devices [89] before being forced to resort to new devices like tunnel transistors [88]. This means that Moore's Law is still alive [132]. However, Dennard's scaling rules, which provided ever decreasing energy per operation ratios, seem to be no longer applicable since the 90 nm generation. Never again was transistor miniaturization able to provide even close to 50% power reductions and 40% frequency increases per generation maintaining power density as in the past.

For these reasons, when in 2004 the limits to conventional air cooling for a typical-size single-core chip, about 130 W, had already been reached at 90 nm, continuing to increase power consumption was no longer a viable option [55]. Chip manufacturers were caught up in a situation in which extracting more ILP was no longer reasonable even with classical transistor energy scaling, and to top it all, they could not even rely on classic transistor scaling to reduce energy consumption any more, resulting in a *power wall* to increasing microprocessor performance. The logical —and maybe the only possible— step to overcome the power wall and keep on increasing performance was to embrace more energy-efficient multi-core designs [64, 143].

The idea behind this shift in design can be explained by means of Pollack's rule, which has remained true throughout Intel processor microarchitectures [26, 150]. Pollack's rule states that in ILP-oriented designs, performance approximately increases proportionally to the square root of the number of transistors used. In other words, assuming the same operating frequency, doubling the transistors used in the design provides a 40% increase in performance (as historic trends confirm), and doubling the performance of a processing core requires spending four times as many transistors in a more aggressive design. To make matters worse, analysis also show that this kind of ILP-based designs increase power consumption *cubically* with performance [46] (assuming no changes in transistor technology and frequency). This is the same as saying that the effective capacitance of the design increases, due to the higher switching activity factor of transistors and circuit capacitance required to dynamically uncover and exploit more ILP (which is also confirmed by historic trends) [96]. Classical transistor scaling used to outweigh these higher energy costs, enabling ever-decreasing energy per operation, although the overall power budgets kept increasing [66].

In comparison, multi-cores posed more favorable prospects: performance can be potentially doubled by just putting two cores side by side, and area and power just approximately double as well (with no increase in the activity factor of transistors or the circuit capacitance). That is, multi-core design has the potential to scale up performance with linear increases in area and power by adding more cores, instead of quadratic area and cubic power budget increases as by exploiting more ILP [26]. By moving back to simpler cores and replicating them, we can reduce the area and power needed to achieve the same performance as with a complex single-processor chip. This reduces the reliance on CMOS technology advances to maintain traditional reductions in energy per operation. The downside of multi-core systems is that all their parallelism is not automatically used to exploit ILP. Now, the programmer (or the compiler) needs to help the processor by explicitly extracting thread-level parallelism (TLP) by means of parallel programming. Nevertheless, this does not mean that multi-cores are not able to take advantage of any ILP. A trade-off between ILP and TLP provides a balance between energy-efficiency and single-thread performance in multi-cores. Basically, the reduction of ILP-extraction logic accounts for the differences in area and power order-of-complexity between multi-core and single-core designs.

Multi-cores provide more potential benefits. For a given area, the potential performance of a multi-core is higher than that of a more complex single core. In addition, the lower power density of multi-cores allows the use of more transistors than in complex single-cores within the same power budget, hence enabling the use of more area to allocate even more cores and boost performance further. Still, multi-cores are expected to run eventually into the same power wall with predicted technology scaling trends [47].

In addition, the linear power and area characteristics of multi-cores just indicate potentials, and their design poses new challenges that may reduce energy-efficiency and need to be addressed from an architectural point of view in order to realize those potentials. Of these challenges, the more pressing ones are cache coherence scalability and network-on-chip power consumption. Today, computer architects are in charge of addressing these challenges to provide a large share of the energy per operation reduction expected in every generation and prevent processor performance from stagnating in today's power constrained scenario.

Multiple fabrication advances have tried to alleviate the situation and put Dennard's scaling back on track. Strained silicon allowed increasing energy efficiency reasonably [167]. Some chip makers resorted to silicon-on-insulator (SOI) for a boost in transistor performance and energy efficiency [162]. However, shrinking transistors involves thinner gate oxide thickness, which at 1.2 nm is made of only about six atomic layers and suffers unacceptable levels of leakage currents due to tunneling [74,78]. Silicon-dioxide insulator was replaced with more dielectric materials such as hafnium dioxide (high-k materials [70]) to improve insulation and reduce leaking, fuelling energy-efficiency. Still, voltages cannot go further down to improve energy-efficiency due to subthreshold leakages as the threshold voltage also needs to go down. More leakage causes greater power dissipation, and despite every effort, classical energy scaling has petered out, leaving us with many transistors available that cannot be used as doing so increases the power consumption of the chip, and a stagnated operating frequency that cannot be scaled up because dynamic power would increase proportionally, killing the two main performance boosters since the inception of microprocessors: more complex designs to extract ILP and higher clock rates [46]. Further breakthroughs at the technology level are needed [42], and technologies such as tri-gate transistors (also known as FinFET or 3D transistors) keep improving energy per operation today [8], but still at rates far lower than past ones. In the meantime, in parallel to this technological fight against physical limitations, it is now time for computer architects to bring forth alternative solutions.

1.2 Shared Memory and Cache Coherence

To exploit the increasing thread-level parallelism provided by CMPs, programmers are required to develop parallel applications that make use of multiple cores. Unfortunately, parallel applications are much more difficult to code than sequential ones. Driven by this difficulty, programmers have widely chosen the simplest parallel programming model available today, *shared memory*, as their preferred paradigm to develop parallel code.

In a shared memory system, cores communicate by means of load and store operations in a shared memory address space. The system ensures that the results of this communication comply with some predetermined rules that make the system predictable and are used by programmers to develop correct parallel applications. These rules make up the *memory consistency model*, which basically determines when cores are able to see —by means of loads— each other's modifications —stores— in the shared memory address space [3]. Most parallel applications and modern operating systems have been developed for shared memory systems because of their convenience [29].

Shared memory would be trivial to implement if load and store operations were directly applied to a physically shared main memory. However, through the years, dynamic memory (DRAM) commonly used for main memory has been optimized for density and price, pushing performance into the background. This trend has created an increasing performance gap between ever-faster cores and memory, resulting in what is considered a *memory wall* that makes direct accesses to memory by the cores impractical [115]. To bridge this gap, fast on-chip cache memories are commonplace today. In CMPs, part of this cache is private to each core, providing small size and fast access —matching core speed—, able to catch most core accesses and keep an illusion of fast main memory. The remaining on-chip cache resources are commonly shared —to some extent—, in an attempt to provide a trade-off between latency and cache utilization. Overall, cache memories make an effective task of preventing slow main memory accesses from ruining system performance.

With the use of private caches in multiprocessors, shared memory becomes more complex to implement because the problem of *incoherence* arises. Potentially, several cores could be working on their own local copies of the content of the same memory location, stored in their private caches, reading and modifying these copies at will with no communication of new values taking place between them. This behavior would break the implicit inter-core communication expected from shared memory systems.

In order to prevent incoherences, an appropriate mechanism is required to automatically propagate newly generated values at the right times. On top of such a mechanism, it is easy to restore the implicit communication and its particular rules expected from the sharing memory system, creating the illusion of a physically shared memory directly accessed by the cores.

The most efficient and common way to carry out inter-core communication and prevent incoherences is by means of *hardware cache coherence protocols* [166]. Cache coherence protocols are in charge of coordinating the low-level interchange of messages between cores so that the private caches are invisible to the programmer, who does not have to explicitly carry out any low-level communications. In particular, they ensure that store operations to shared memory are eventually made visible to all cores, and that store operations to the same location appear to be seen in the same order by all cores (although these conditions may be relaxed by some memory models, as will be explained later).

Simply stated, cache coherence makes sure that every core always accesses the most current value of each data. This can be done in several ways, from letting every core observe all memory operations to carefully orchestrating communications between those particular cores affected by each memory operation. The latter is more scalable, as the traffic requirements are smaller and it can make efficient use of point-to-point interconnects. On the other hand, that option introduces extra storage requirements like the presence of a directory to track the contents of private caches to enable such fine-grain communication.

In general, cache coherence protocols preserve one invariant: at any given time, either only one core may be writing the data, or from zero to many of them may be reading it, but no core should read data while another is writing it. This is typically done at a granularity of a memory block, usually comprising 64 bytes. Upon the update of a memory block by a core, the current readers of the block (if any) must be warned to prevent them from accessing stale values in the future. This can be done either by updating their local copies of the block (providing them with immediate access to the new value) or by invalidating their copies of the block (forcing them to retrieve the new value when they next access the block). The latter is often the preferred option due to its smaller communication footprint, as many updates by a single core may take place prior to new accesses by other cores.

The cache coherence protocol only enforces ordering between operations on individual memory locations. On top of it, the shared memory system requires a memory consistency model to ensure that the execution results comply with a valid ordering between all operations to *all* memory locations. In the simplest memory consistency model, *sequential consistency*, any total ordering is valid in which the operations of each core appear in program order, regardless of the interleaving between operations of different cores. Intuitively, sequential consistency is automatically guaranteed by a system with cache coherence if cores just issue one memory operation to the memory system at a time, in program order. However, this model is very restrictive for performance. Alternatively, the particular memory consistency model may relax some conditions in order to enable hardware optimizations to boost performance —such as issuing more than one simultaneous memory operation per core, or issuing them in a different order to program order, resulting in different cores perceiving different global orderings. In that case, the programmer should deal with the —also well defined— new rules when programming their applications, resorting to special instructions typically provided to force orderings between operations when necessary.

Popular libraries such as pthreads [136] or OpenMP [40] are easily and efficiently implemented on top of shared memory systems, providing convenient higher-level concurrent execution and synchronization mechanisms to programmers to write efficient parallel applications.

Compared to alternative models, such as message passing [67] or incoherent scratchpad memories [11], that require explicit low-level communication by the programmer, with shared memory programmers just need to follow the shared memory rules —or use higher level libraries— and focus on finding ways to efficiently parallelize applications without worrying about communication details. This and the difficulties associated with porting most current software developed for shared memory to other parallel programming models seem a strong-enough guarantee that cache coherence mechanisms will remain supported in future CMPs. The drawback of such a simple programming model is that the cost of supporting hardware cache coherence protocols needed for efficient shared memory may become a barrier to scaling out CMPs.

Scaling coherent cache hierarchies for the envisioned chip multiprocessors that integrate large numbers of cores (e.g., hundreds or thousands) is problematic. The lack of scalability of existing coherence mechanisms in terms of area, traffic and energy-efficiency may limit the applicability of cache coherence to future CMPs.

There are two main approaches to keeping coherence: snoopy based cache-coherence protocols and directory-based cache coherence protocols. Snoopy-based cache coherence protocols [65] cannot scale beyond a few cores due to their reliance on broadcast, which causes large volumes of network traffic and a high number of energy-consuming cache lookups [97].

Large systems usually rely on directory-based cache coherence protocols [31]. The directory keeps information about which cores are using each cache line, enabling efficient point-to-point coherency traffic that uses scalable networks, such as meshes or tori, at the expense of requiring extra memory for storing the sharing information.

Unfortunately, designing a scalable directory for large core counts is not easy, especially if the directory stores exact sharing information, which is necessary for minimizing network traffic. Co-locating the directory information with the tags of the shared level of the cache hierarchy works well for small systems [9, 34], but incurs an overhead per cache entry proportional to the number of cores, making it unsuitable for large systems. Sparse directories [69] use less area by tracking the contents of the private caches in a separate small directory cache, but their per-core overhead remains proportional to the number of cores. In addition, the limited capacity of sparse directory caches introduces harmful coherence invalidations of active private-cache blocks [51]. Unlike sparse directories, duplicate-tag directories' per-core overhead is proportional to the cache line tag size and insensitive to the number of cores, and they do not introduce coherence invalidations [12]. Unfortunately, their associativity is proportional to the number of cores, making them impractical for large systems for energy-efficiency reasons.

A number of proposals try to reduce the size of the directory's sharing code and the frequency of directory-induced invalidations. These usually imply an increase in the complexity, latency and energy consumption of the directory circuitry and the cache coherence protocol. For instance, hierarchical directories [125] have been proposed as a scalable alternative to flat directories. By distributing the sharing information in several levels, their per-core overhead is proportional to the k-th root of the number of cores (where k is the number of levels of the hierarchy). However, the complex management of the distributed sharing information makes these protocols difficult and costly to validate and implement in real systems.

Inexact sharing codes are another way of reducing the directory overhead [2, 4, 69, 110]. However, code inexactitude causes superfluous coherence actions, increasing network traffic and execution time and compromising scalability. In addition, those inexact codes with the smallest overheads (e.g., limited pointers with logarithmic overhead) suffer from superfluous coherence actions the most, making them impractical for large core counts [125].

To this day, computer architects remain struggling to provide efficient designs to support cache coherence with lower sharing-information area overhead, fewer extra coherence invalidations and lower traffic requirements, to enable cache-coherent CMPs with larger core counts in the current energy-constrained scenario.

1.3 Networks-on-Chip and Cache Organizations

With the advent of CMPs, and especially as the number of cores increased in them, it became necessary to integrate a network for communications within the chip, known as network-on-chip or simply NoC [19]. This NoC must be flexible enough to efficiently carry out core accesses to the shared cache resources, to perform communications between cores, and to reach the memory controllers for off-chip DRAM accesses and I/O communications. Usually, electronic meshes have been frequently proposed as simple, reliable and efficient networks for point-to-point communication [79, 84, 152, 157, 171], adequate for scalable cache coherence protocols, once simpler interconnects such as buses and crossbars were insufficient to carry all CMP traffic due to bandwidth limitations. Meshes are more scalable than these networks, as their aggregate bandwidth increases proportionally with the number of cores.

As a first approximation, on-chip traffic should grow linearly with core count, matching the aggregate bandwidth of typical electronic NoCs and maintaining multi-core's potential to add cores with linear power requirements. However, as the core count rises, so do the relative distances between nodes on current electronic NoCs, measured as the number of intermediate retransmission points. For instance, communicating opposite corners of a chip in a 4×4 electronic mesh requires 6 retransmissions, a figure that goes up to 14 retransmissions when scaling the mesh up to 8×8 nodes. This creates two serious interrelated problems. First, for current electronic NoC scaling, the pace at which bandwidth increases (proportional to core count) is not enough to sustain the traffic growth (proportional to core count times the growing average number of retransmission points). Second, the energy consumption of such traffic is growing quickly under current NoC scaling, affecting energy per operation negatively.

In a time when limited transistor energy scaling makes it increasingly difficult to accommodate extra logic in a limited power envelope, current NoC scaling trends make logic components look cheap when compared to the cost of the communications required to supply data to those components, which gives an idea of the magnitude of the NoC problem. For this reason, it is of paramount importance to prevent retransmissions in the NoC, be it by rethinking the placement of data to increase proximity or by considering new transmission technologies with better scaling properties. Otherwise, NoC energy will become an insurmountable barrier to scaling out CMPs, taking up all the power budget of the chip, as some analysis have already reported [27].

In particular, the on-chip cache hierarchy plays a key role in data proximity, becoming fundamental for good energy-efficiency. It has been estimated that chip area spent for cache uses around 10 times less energy than core logic for typical activity factors. Hence, adding cache resources to the chip to improve overall system performance is more energy-efficient than increasing the area occupied by logic components, as cache enables the use of more transistors with the same power budget thanks to its lower power density, and in addition it helps alleviate the *memory wall* by preventing costly main memory accesses [115]. This results in increasingly large amounts of cache being commonly integrated in chips nowadays, a trend that is expected to go on fuelled by power constraints [14,85]. So many cache resources must be organized in the most energy-efficient way, especially in a CMP environment in which the cache organization must promote data proximity to counter the power consumed by the increasing number of NoC retransmissions when accessing remote cache resources.

Usually, several levels make up the cache hierarchy, typically two or three. Part of the cache resources are employed in creating first-level private caches next to each core. These private caches must be small enough to provide fast access and large enough to catch most accesses. As we move to higher levels, caches may start to be shared among cores, but not necessarily.

Particularly, the last-level cache (LLC) is physically distributed among the cores, and different LLC organizations are possible to avoid costly off-chip accesses and to optimize on-chip communication and cache utilization. These organizations range from a completely shared LLC [80,95,100,105], in which all the cache banks of the chip can be accessed by any core, to private LLCs [1,7], in which each core only accesses its own private cache bank, with intermediate partially shared cache organizations in between [75,87,135], in which cores share their cache resources forming clusters.

Logically, increasing the degree of sharing also increases the average distance that needs to be travelled by the messages, because some of the LLC banks will have to be further from the core. Hence, as the number of cores in CMPs increases, totally shared organizations become less attractive in this regard. However, some degree of sharing is still desirable because it improves LLC utilization. This is because shared caches allocate a single copy of each shared data for all cores, contrary to private caches which replicate shared data in the private caches of every core that accesses these data. This replication increases cache pressure and it results in additional LLC misses, which in turn increase off-chip traffic. Since off-chip bandwidth increases more slowly than the number of cores in a CMP [154], avoiding LLC misses becomes even more important as the number of cores increases. Hence, partially shared organizations provide a good trade-off between the fast latency of private caches and the improved capacity utilization of shared caches, and computer architects must research novel organizations with better data-proximity properties.

An orthogonal way of tackling the problem of NoC scaling is by using disruptive technologies with noticeably higher energy efficiency than electronics,

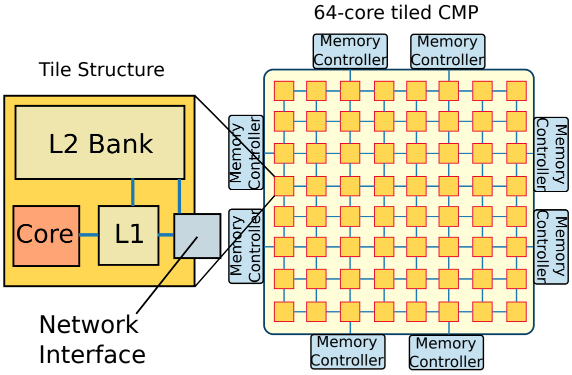


Figure 1.1: Diagram of a 64-core tiled CMP.

for example by avoiding retransmissions. An example of such technologies is photonics, which introduces light-based transmission resources in the chip [68,90]. These resources commonly comprise waveguides to drive the light injected by an on- or off-chip laser, and microring resonators and photodetectors to perform, optionally route, and receive optical transmissions.

Silicon photonics has been demonstrated to enable light-speed transmissions that are near insensitive to distance in terms of latency and energy consumption. This technology is free of the energy problems of the increasing number of retransmissions suffered by electronic NoCs, with better energy and latency features. With photonics, communications potentially require no retransmissions independently of the number of intermediate nodes. When scaling up CMPs, a smart integration of photonics on-chip gains interest notably.

Ideally, the best features of both transmission technologies, electronics and photonics, should be exploited in CMPs. For example, an ordinary electrical mesh and a photonic network can cooperate to minimize overall latency and energy consumption. In particular, electronic-based communications are still interesting as they have good bandwidth, latency, power and layout-flexibility for short-range communications. Today, computer architects are still in preliminary phases of researching the best ways to integrate photonics in CMP architectures, and currently there are no operating photonic NoCs widely available.

1.4 Cache-Coherent Tiled CMPs

Putting it all together, cache-coherent tiled CMPs [18,37,52,81,149,152,171] arise as the archetypal chip of the current situation of computing in most market segments.

Tiled CMPs are made up of (almost) identical tiles that are replicated. Each tile contains a processing core, private cache resources —typically dedicated data and instruction L1 caches—, a portion of the shared cache resources —for which a myriad of organizations are possible—, storage and logic to enforce scalable cache coherence, and a network interface to communicate with other parts of the chip. A NoC, typically an electric mesh, is deployed in the chip to interconnect all tiles. This design embodies the main challenges of increasing performance by scaling out CMPs. For this reason, tiled CMPs were the architecture assumed throughout this thesis. Figure 1.1 shows a schematic layout of a 64-core tiled CMP, which is similar to the Tilera Tile64 [18] for its conceptual representativeness. We assume two levels of cache throughout this thesis, without loss of generality.

Other architectures exist that are gaining momentum thanks to their energy efficiency, such as throughput oriented GPUs [144], specific circuits (ASICs, ASIPs [99]), or even CMPs with different layouts —non-tiled—. If anything, we consider these architectures complementary to tiled CMPs, and they are starting to be integrated in the same die (e.g., heterogeneous chips [28,73]). We believe that these architectures do not strip tiled CMPs from any representativeness of the challenges when scaling out CMPs to large core counts.

In addition, the design of tiled CMPs simplifies the construction of large chips, which can be done by simply adding more tiles. The costs of designing and validating more powerful chips are noticeably reduced, especially if compared to the past trend of developing novel microarchitectures to exploit ILP.

For cache coherence, we assume directory-based invalidation protocols using MOESI states, as they use point-to-point communication and provide good performance and reasonable scalability for our baseline architectures.

Each proposal described in this thesis assumes this design as a baseline but with particular variations of its features, depending on the specific requirements of the particular scenario on which the proposal is applied. The details of these specific features, which include the particular LLC organization, the number of cores per tile and in the chip, the resources required by cache coherence, the operation of the cache coherence protocol and the NoC deployed, are specified in the corresponding chapters of this thesis.

1.5 Motivation

To allow the progress of multiprocessors, new energy-efficient designs are required to overcome the barriers to scaling out CMPs to core counts in the hundreds. In this thesis we tackle two of the main multi-core-specific challenges, namely cache coherence scalability and NoC power consumption, which may clearly become a limiting factor in reducing the energy per operation of larger CMPs. We propose mechanisms to reduce the impact of these issues. In particular, the following opportunities motivate this thesis:

- The cache coherence protocol coordinates on-chip communication and can be optimized to improve data proximity, also allowing reduced size for exact tracking codes, improving the scalability of coherence.

- Cache organizations can be optimized to improve data proximity by reorganizing the mappings regarding which cores access which last-level cache banks, shortening distances and reducing the number of retransmissions in the NoC.

- The shared cache resources of the chip can be used more efficiently to store data and sharing information, improving cache coherence scalability.

- Photonics and electronics can be combined to exploit the best features of each technology, increasing NoC efficiency with respect to using just one technology, and adequate management policies are necessary to do it efficiently.

Microprocessor energy-efficiency can be improved by means of trade-offs between performance and power dissipation, that is, by forgoing some performance for a relatively higher reduction in power consumption, or alternatively, by increasing performance for a relatively smaller power consumption increase, reducing overall energy per operation in both cases. Such trade-offs are paramount when setting the physical parameters of microprocessors like the voltages and frequencies that yield the best energy per operation ratio for current technology (e.g., subthreshold CMOS [119,181]). Also, architecture designs often search for settings that produce the best performance-power trade-offs to reduce energy per operation. However, truly energy-efficient designs are based on improving the way the microprocessor operates. By performing the same task more efficiently, doing less (unnecessary) work and using less resources, less power is required during the process and it is completed faster, with multiplicative effects on energy-efficiency. We have tried to design proposals that operate in this way to the maximum possible extent.

1.6 Thesis Contributions

The following subsections summarize the contributions of this thesis.

1.6.1 Multiple-Area Cache Coherence Protocols

We propose a family of cache coherence protocols that statically divide the chip in areas. Coherence is maintained per area, and pointers link the areas to keep a single coherence domain, reducing the directory overhead to save energy. This reduction is especially significant with large core counts (e.g., 93% overhead reduction compared to a full-map directory for 1024 cores and just 4 areas). The coherence protocol dynamically selects one node per area to act as *provider* of the data for cache misses in that area. A prediction mechanism is used by nodes to reach the provider within their areas, preventing directory indirection and reducing the average cache miss latency and network traffic. Compared to a highly-optimized full-map bit-vector directory implementation, these protocols reduce the leakage power consumption by 54% and the dynamic power consumption of the caches and the network-on-chip by up to 38% for a 64-tile chip multiprocessor with 4 areas, showing no performance degradation. They were proposed in the context of server consolidation by means of virtualization, but were proven equally applicable to multiprogrammed workloads and parallel applications.

1.6.2 Distance-Aware Partially Shared Cache Organizations

The higher the core count in a chip, the more interesting partially-shared designs become for last-level caches (LLC) [87]. In a traditional partially-shared cache, cores share their LLC banks in clusters, a compromise design that causes less memory accesses than private LLCs, and less network traffic and lower hit-latency than shared LLCs. The number of cores that form each cluster, sharing their LLC banks, is the *sharing degree* of the cache. We propose DAPSCO as an optimization to traditional partially-shared caches, based on the observation that clustering the LLC banks is not the most efficient organization. For example, a core in a corner of the cluster is on average farther away from the cluster LLC banks than a core in the center. DAPSCO uses a more efficient core-bank mapping in which no clusters exist and each core accesses its surrounding LLC banks, giving every core the impression that it is located in the center of a cluster, minimizing the average distance to the LLC. DAPSCO's mapping holds the same desirable properties as traditional clustered organizations: LLC banks store subsets of the memory space, and each core accesses one bank for each subset to reach all the memory space. We have generated and evaluated DAPSCO configurations for core counts from 64 to 512 with varying sharing degrees and network topologies. For instance, DAPSCO reduces the average number of links to reach the LLC by 33% in a 512-core CMP with a sharing degree of 128, reducing the latency of LLC accesses and energy usage of the network accordingly. We tested DAPSCO on a 64-core CMP to confirm its theoretical advantages. Moreover, DAPSCO introduces almost no overhead, as it just requires small changes in combinational circuits already present in the chip.

1.6.3 In-Cache Coherence Information

ICCI is a new cache organization that leverages shared cache resources and flat coherence protocols to provide inexpensive hardware cache coherence for large core counts (e.g., 512), achieving execution times close to a non-scalable sparse directory while noticeably reducing the energy consumption of the memory system. Very simple changes in the system with respect to traditional bit-vector directories are enough to implement ICCI. Moreover, ICCI does not introduce any storage overhead with respect to a broadcast-based protocol, yet it provides large storage space for coherence information. ICCI makes more efficient use of cache resources by dynamically allowing last-level cache entries to store blocks or sharing codes, resulting in a leaner design than when using dedicated structures. Moreover, ICCI design guarantees a negligible amount of directory-induced invalidations of active blocks, compared to the potential high amount of these in traditional schemes. Results for a 512-core CMP show that ICCI reduces the energy consumption of the memory system by up to 48% compared to a tag-embedded directory, and up to 8% compared to the more-complex state-of-the-art Scalable Coherence Directory [159] which ICCI also outperforms in execution time, resulting in a multiplicative effect on energy-efficiency. In addition, ICCI can be used in combination with elaborated sharing codes to apply it to extremely large core counts.

1.6.4 Management Policies for Exploiting Hybrid Photonic-Electronic Networks

We propose dynamic policies for efficient management of hybrid photonic-electronic networks. Unlike electronics, photonics is efficient for long distance communications in a chip as light requires no retransmissions to reach its destination. On the other hand, electronics has good capabilities for communicating close elements. So far, hybrid networks used static policies to decide whether to use photonics or electronics for the transmission of a message. We have developed policies that use real-time information to make this decision at a message granularity. We tested a number of these policies on an electronic mesh combined with a photonic network using both real applications and synthetic traffic. A small FlexiShare-based ring was used as a near-future scenario, and large Corona and Firefly designs were tested as long-term networks for 256 cores. By combining information such as the size of the message, the time already waited for photonic transmission and the distance between endpoints, we developed a policy that obtains large energy and performance benefits over alternative ways to manage the hybrid network. This policy attains a throughput for the hybrid network up to 38% larger than the sum of the throughputs of the parts (the photonic network and the electronic mesh) in a 64-endpoint scenario. It also achieves an average latency lower than any of the sub-networks by using them smartly (i.e., short distance transmissions using the mesh, and long distance ones using photonics).

1.7 Publications

Parts of the research performed for this thesis have been published in relevant conferences and journal publications or are currently under review. We outline briefly these works detailing which chapter of the thesis is covered by them.

- In the 5th Annual Workshop on Modelling, Benchmarking and Simulation [56], held in conjunction with the 36th International Symposium on Computer Architecture in 2009, we introduced Virtual-GEMS, the simulation infrastructure used along this thesis that enables the modeling of consolidated servers with ease.

- In the 22nd International Symposium on Computer Architecture and High Performance Computing [61], we presented our first approach to cache coherence protocols for server consolidation by analyzing the performance of state-of-the-art proposals, designing solutions to their flaws, and evaluating the impact of techniques such as memory deduplication in the context of cache coherence protocols adapted to virtualized environments.

- In the 40th International Conference on Parallel Processing [62], we introduced our idea about Multiple-Area Cache Coherence Protocols in the context of server consolidation. We gave some preliminary results and stated the convenience of having a single coherence domain composed of different physical cores. This paper was presented in 2010, and it corresponds to Chapter 3.

- We followed up our previous work with a paper that, from a theoretical point of view, analyzed the implications of low-overhead cache coherence and broadcast management with respect to protocol correctness and provided solutions to some issues posed by this scenario. This paper is currently under review [59].
- In the January 2012 issue of ACM Transactions on Architecture and Code Optimization [63], we introduced our proposal on Distance-Aware Partially Shared Cache Organizations, showing the advantages of enabling each core to access the LLC banks surrounding it. This work was presented in January 2012, in the 7th International Conference on High-Performance and Embedded Architectures and Compilers, and it corresponds to Chapter 4.
- Our work on In-Cache Coherence Information, corresponding to Chapter 5, is currently under review [58].
- Our work on Dynamic Policies for Hybrid Photonic-Electronic NoCs, corresponding to Chapter 6, was carried out during a stay at the Università degli Studi di Siena that took place from February through July 2012 under the supervision of Sandro Bartolini. This work is currently under review [60].

1.8 Thesis Organization

The rest of this thesis is organized as follows. First, Chapter 2 gives some background on cache coherence and photonic transmission that is necessary to understand the remaining chapters of this thesis. Then, one chapter is dedicated to each of the research paths explored in this thesis (Chapters 3, 4, 5 and 6). These four chapters are structured in the same way. A brief introduction provides a global view of the problem approached and the solution proposed in the chapter. This is followed by a section giving background on the current situation regarding that particular topic, including those proposals used as a comparison point. Then, the main section of the chapter describes our proposal to improve the energy efficiency of CMPs. Later, this proposal is evaluated against state-of-the-art alternatives and the results are analyzed. A section on related work puts our proposal into context with other relevant proposals on the topic. Finally, a section summarizing our findings concludes the chapter. These chapters are mostly self-contained in order to facilitate their independent reading. To conclude, Chapter 7 summarizes the contributions of this thesis and poses some promising future paths of research that may follow up from our work.

Chapter 2

Background

In this chapter, we describe some fundamental concepts required to understand the proposals in this thesis, including the operation of cache coherence mechanisms used in the rest of the chapters, as well as the foundations of data transmission in photonic networks.

2.1 Cache Coherence Basics

This section describes some basic concepts of cache coherence that are necessary to understand the proposals presented in the following chapters. Since it is possible to find different definitions for some of these concepts in the literature, this section also has the purpose of setting their precise meaning in this thesis to prevent any ambiguity.

Cache coherence plays a key role in building the memory consistency models of shared memory systems. Cache coherence consists of maintaining a single view of the contents of each memory location for all cores to prevent incoherences, that is, to prevent cores from reading stale values from their private caches that would break the memory consistency model. For this, a cache coherence protocol interchanges messages to set the logical order in which cores write new values in a memory location, invalidate old values and communicate the new values to other cores that read them.

We focus on describing the operation of a cache coherence protocol based on a MOESI scheme that uses a directory for storing sharing information. This coherence protocol sends messages only to the precise nodes taking part in a coherence transaction, taking advantage of the features of unordered point-to-point networks, such as meshes. This is the base coherence architecture assumed by all proposals in this thesis.

For our purposes, in this explanation we divide the memory hierarchy in two parts, without loss of generality in the description of the cache coherence mechanisms. We differentiate between private caches, among which it is necessary to maintain coherence, and the remaining upper levels (e.g., any shared cache levels and main memory). We assume that the private caches correspond to the private L1 caches of the cores and the upper levels correspond to a shared last-level cache (L2 cache) and main memory.

It is convenient to start by describing the meaning of each of the states that correspond to the letters in MOESI. These are the states in which a memory block may be stored in each of the private caches. Note that just one copy of the memory block may be stored in each private cache, in one and only one of the following states (although several copies may exist in several caches and in different states):

M. Modified. The value of the memory block is modified in the private cache (i.e., the block is *dirty*) and no other valid copy of the block exists in the system, including in other private caches and in the upper levels of the memory hierarchy. The core associated to the private cache has read and write permissions on the block.

O. Owned. The block may be dirty in the private cache containing it in O state. If so, its value is not updated in the upper levels. Otherwise, the block is *clean* (i.e., its value is updated in the upper levels). In addition, other valid copies of the block may exist in other private caches, in any case. The core associated to the private cache only has read permission on the block.

E. Exclusive. The block is stored in the private cache containing it in E state and in no other private cache. In addition, the block is updated in the upper levels (i.e., the block is clean). Even so, if a copy of the block is required by another core, this private cache must provide it, as the local core has permission to modify the block any time (transitioning immediately to M state) without warning the upper levels. The core associated to the private cache has read and write permissions on the block.

S. Shared. The block may be present in other private caches. The core associated to the private cache only has read permission.

Table 2.1: Allowed combinations of states in the private caches for a memory block.

	M	**O**	**E**	**S**	**I**
M					✓
O				✓	✓
E					✓
S		✓		✓	✓
I	✓	✓	✓	✓	✓

I. Invalid. The value of the block may be stale and therefore the block is not valid. The core has no access permissions on the block. We assume that *invalid* and *not present* (i.e., no copy of the block exists in the private cache) are equivalent for our purposes.

Copies of the block in different states may coexist in several private caches. The combination of states must not break the coherence invariant that requires that at any given time, either only one node may write to a memory location, or from zero to many may read it [166]. The allowed combinations of states are shown in Table 2.1. To ensure that when one node can write the block no other node can read it, the E and M states cannot coexist with any other state but I in the private caches (and only one private cache may store the block in M or E states). To allow from zero to many nodes to read the block (and ensure that none can write it), the O, S and I states can coexist freely, except that only one private cache may store the block in O state for practical reasons. The private cache storing the block in O state has the function of keeping the updated value of the block, which may be stale in the upper levels, and update the upper levels upon eviction when the block is dirty to prevent the loss of its value. In addition, this private cache also provides a copy of the block to other nodes when required.

When a core requests a read or write operation, it will be able to perform it directly in the cache if the block is stored there with the necessary permissions. In a well performing system, this will happen most times. However, there are occasions in which a memory operation cannot be satisfied by the local private cache (because the data is not available there or the permissions are not enough), resulting in a *cache miss*. In this case, steps must be taken according to the cache coherence protocol to complete the memory operation of the core, taking into account the state of the memory block in other cores and the kind of request (load or store). Next, we define the actors that are involved in private-cache misses.

Requestor. It is the core that suffered the private-cache miss due to the absence of a valid copy of the block (i.e., the block is in I state) or lack of permissions (e.g., upon a store request to a block in a state different to M or E).

Home. It is the node in which the coherence information (directory) is located, required to determine the actions to take to resolve the cache miss of the requestor. In this thesis, we assume that the home node is one of the LLC banks, following a NUCA distribution of memory addresses.

Owner. It is the node that holds the current version of the block (do not mistake with the owned state). It is a private cache when it stores the block in M, E or O state, and the upper levels otherwise (an L2 cache or the memory).

Sharer. It is any private cache that contains a copy of the block in S state. They need to be invalidated to preserve the coherence invariant upon a cache miss caused by a store operation.

Notice that the remaining nodes (e.g., other private caches) are not involved in the resolution of a cache miss.

The cache coherence protocol has to intervene also in the resolution of evictions of blocks caused by private-cache conflicts (i.e., blocks contending for the same entries of the private caches).

Upon a cache miss or an eviction requiring coherence actions, the requestor sends a network message containing the request to the home node. These requests are shown in Table 2.2, putting them in relation with the states of the block in the requestor (MOESI) and the events that, in combination, generate the requests. The check marks indicate those cases in which the events can be completed locally, without sending a request to the home node. These are cases in which the core has a copy of the block in a state containing enough permissions to carry out a memory operation or a replacement takes place in I state, which enables the core to just discard that block. Otherwise, load and store requests are sent to the home node upon memory operations (i.e., when the state of the block in the private cache is I for load operations and different to E and M for store operations). Those cases in which the block is already present in the private cache but a memory operation requires more permissions are marked with an asterisk. For simplicity, we reuse the name *store* for the request in this case, which is typically known as *upgrade* in the literature [166].

Upon a block eviction, a message is sent to the home node to update the directory information (C.Rep message). In addition, if the evicted copy of the

Table 2.2: Messages to home node generated upon events in the requestor.

	State of the Block				
Event	**M**	**O**	**E**	**S**	**I**
Load	✓	✓	✓	✓	Load
Store	✓	Store*	✓	Store*	Store
Eviction	D.Rep	D.Rep/C.Rep	C.Rep	C.Rep	✓

block is in M state or O state and dirty, the value of the upper levels needs to be updated or the current value of the block would be lost (D.Rep message instead of C.Rep message, involving a write-back operation).[1]

Upon the reception of a request, the home node makes a decision based on the directory information that it maintains for the block. Table 2.3 relates the possible states in the home node and the types of requests, describing how the request is resolved, possibly transitioning the state of the home node for the block and/or carrying out some actions. We are assuming three states in the home node that are based on the state of the block in the private caches (*All I* when no copies exist in the private caches; *M,O,E* when there is a private-cache storing the block in M, O or E states; and *S* when one or more private caches store the block in S state and no other state). If possible, the home node resolves cache misses locally by retrieving the block from the upper levels of the hierarchy and answering back with the data (reply action). Otherwise, the load or store request is forwarded to the owner private cache, which is storing the block in E, M or O state (forward action), incurring *directory indirection*. Then, the owner private cache replies to the requestor with an updated copy of the block. Upon a store request, the home node sends invalidations to every sharer of the block (if there are any). The cases in which invalidations may be necessary are marked with an asterisk in the table. We assume that the node carrying out the store operation receives the acknowledgement messages from the invalidated sharers. Notice that the state transitions caused by replacements keep the state of the block in the home node in accordance with the state of the block in the private caches (e.g., the S state in the home node is reached when the owner private-cache evicts the block and sharers remain in the chip).

[1] A popular optimization consists of carrying out silent evictions (i.e., no message is sent to the home node) when a block is in S state or even in other states when it is clean. However, this optimization can overload the directory with stale sharing information for blocks that were

Table 2.3: State transitions and actions upon the reception of requests in the home node.

	State		
Request	**All I**	**M,O,E**	**S**
Load	M,O,E/reply	–/forward	–/reply
Store	M,O,E/reply	–/forward*	M,O,E/reply*
C.Rep/D.Rep (last copy)	×	All I/–	All I/–
C.Rep/D.Rep from owner (other copies remain)	×	S/–	×
C.Rep from sharer (other copies remain)	×	–/–	–/–

Table 2.4: State transitions in the private caches (assuming atomic transactions).

	Current State				
Event	**M**	**O**	**E**	**S**	**I**
Local read with all in I	-	-	-	-	E
Local read	M	O	E	S	S
Remote read	O*	O*	O*	S	I
Local store	M	M	M	M	M
Remote store	I*	I*	I*	I	I
Eviction	I	I	I	I	I

To conclude with the description of the operation of the protocol, we show all possible state transitions in the private caches in Table 2.4. We differentiate the following events that may cause state transitions in one or more private caches: load by the core, remote load (a load operation initiated by another core), store by the core, remote store (a store operation initiated by another core), eviction, and finally a read by the core with every private cache in I state. Those states marked with an asterisk involve that the private cache transitioning its state sends the data block to the requestor, which is enabled by the forward actions of Table 2.3 carried out by the home node.

The state machine described by Table 2.4 can be synthesized in a number of rules that result more intuitive in describing the operation of the MOESI cache coherence protocol. These rules are the following:

- A store operation makes the block be in M state in the private cache of the requestor, always, regardless of the previous state.

replaced silently by all sharers. It will be indicated in the appropriate parts of this thesis if this optimization is used.

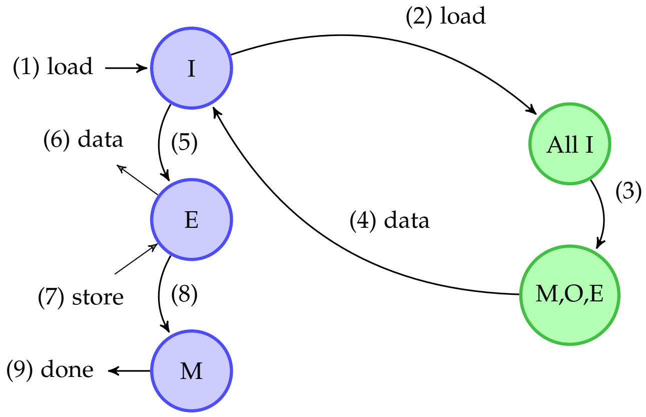


Figure 2.1: Resolution of consecutive load (1) and store (7) operations to a block in I state in all private caches. Sequence of network messages (2,4) and state transitions (3,5,8) in the requestor (blue) and the home node (green) to resolve the load miss (6) and the subsequent store request (9).

- A remote store makes all private caches (except the requestor's) have the block in I state, regardless of their previous state.
- A load operation with the block in I state causes a transition of the block to S state, except when all private caches have the block in I state, which makes the requestor's private cache have the block in E state.
- A remote read to a block in E or M states causes a transition of the block in the private cache to O state.
- The remaining combinations of events and states cause no changes in the state of the block in a private cache.

Tables 2.2, 2.3 and 2.4 combined describe the complete operation of the cache coherence protocol. When one of the local events of Table 2.4 requiring a state transition takes place, it involves that one of the requests of Table 2.2 is sent from the requestor to the home node, where based on the directory information (see Table 2.3), messages are sent just to those private caches that also must transition their state (e.g., every private-cache containing the block in a state different to I upon a write request) or must provide the block (asterisks in Table 2.4), causing remote events. Such coherence protocol prevents the execution of unnecessary actions in the rest of private caches (usually the great majority), whose states remain unchanged.

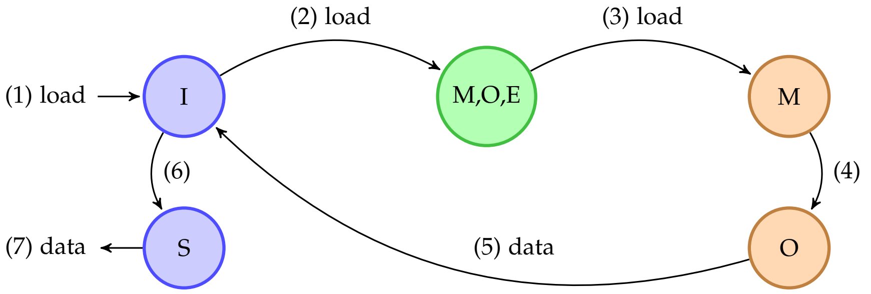


Figure 2.2: Resolution of a load miss (1) for which an owner private cache exists. Sequence of network messages (2,3,5) and state transitions (4,6) involving the requestor (blue), the home node (green) and the owner private cache (orange) to resolve the load miss (7).

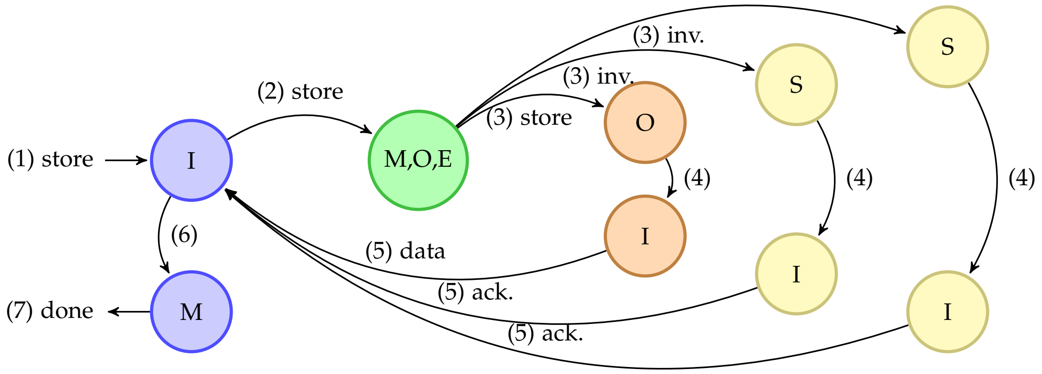


Figure 2.3: Resolution of a store miss (1) to a block for which an owner private cache and sharers exist. Sequence of network messages (2,3,5) and state transitions (4,6) involving the requestor (blue), the home node (green), the owner private cache (orange) and sharers (yellow) to resolve the store miss (7).

Figures 2.1, 2.2, 2.3 describe three examples that illustrate the detailed operation of the cache coherence protocol, showing the sequence of interactions between the requestor, the home node, the owner node and the sharers necessary to resolve different types of cache misses.

The state transitions of Table 2.4 are not atomic when using an unordered point-to-point network such as the mesh assumed in this thesis, as a sequence of messages and events is necessary to complete the transition. This makes possible the appearance of races between conflicting transitions regarding the same memory block. To preserve the coherence invariant, these transitions are serialized by an ordering point (the home node) and many transient states are used to record the particular sequence of events towards the final state of the current transition, simulating atomicity. Although these transient states are implemented in all our proposals of directory-based cache coherence protocols, they are not discussed in more depth, except in the cases that they are particularly relevant to the proposal being discussed.

As an aside, other alternative state schemes to MOESI exist, such as MESI and MSI. They are easy to define by putting them in relation to MOESI. In MESI, the O state is not present. The table equivalent to Table 2.4 for MESI results of removing the column of the O state and replacing the O state with S wherever it appears in the remaining columns. The absence of O requires the upper levels to be updated when a block becomes sharer (S state) in the private caches. In MSI, the E state does not exist either. The table equivalent to Table 2.4 for MSI results of removing the columns of the E and O states and replacing these states with S wherever they appear in the remaining columns. The absence of the E state also makes it necessary to issue two requests to the home node to satisfy the memory operations in the example of Figure 2.1, the first request for the load operation (transitioning to S state) and the second request for the store operation (transitioning to M state). With MOESI, just one request is necessary, accelerating this common sequence of state transitions for private blocks.

2.2 Photonics Basics

In this section, we introduce some basic concepts about photonics that are necessary to understand the operation of the networks under study in Chapter 6. In this thesis, photonic elements are used to tackle the scalability issues of NoCs from an architectural point of view. Therefore, the descriptions provided here are oriented to illustrating the way these elements operate and what functions they fulfill at the architecture level, rather than the detailed physical characteristics that make their operation possible.

The elements necessary to build the networks under study are waveguides, a light source, modulators and detectors. We describe them next.

Silicon waveguides are on-chip channels that carry light across the chip. This light is used to modulate information to carry out communication. A high refractive index difference between the waveguide core and the cladding ensures that light is driven efficiently, preventing losses and crosstalk between close waveguides, as well as allowing bends in the waveguide to enable flexibility in the design of the network. Waveguides are able to carry signals over longer distances at higher bitrates with lower losses than electrical wires. They enable lower delays and prevent the need for retransmissions.

As a **light source**, lasers are typically used (on or off chip) to inject light into the waveguide. WDM (Wavelength Division Multiplexing) consists of using light with different frequencies or wavelengths to transmit information in several channels simultaneously, with one bit of width for each channel, corresponding to a particular wavelength or *carrier*.

Microring resonators have multiple uses, such as modulator, filter or switch, being the key enabler of a myriad of nanophotonic interconnect designs. A generic ring resonator is made of a looped optical waveguide [24]. Very small microring resonators are enabled by silicon (e.g., with a radius of 1.5 µm [177]), playing a major role in the success of photonics. For our purposes, we use them as **modulators** and as filters that in combination with a photodiode make up a **detector**.

When working as modulators and filters, microring resonators are placed near the waveguide that carries the light on which they act. Their operation is based on resonance, which occurs when the optical path length of the microring resonator is exactly a whole number of wavelengths of an optical signal transmitted on the adjacent waveguide. This makes a microring resonator sensitive to a certain set of wavelengths that make the microring be on-resonance. Multiple carrier signals, with a different wavelength each, are transmitted along the signal waveguide, by means of light containing these wavelengths injected by the light source. If the wavelength of a carrier matches the resonant wavelength of a microring, the microring is on-resonance and the light corresponding to that wavelength is coupled into the microring and removed from the waveguide as a result. As modulators, microring resonators are electrically actuated devices. When on resonance, the microring destroys the light circulating in the waveguide corresponding to its resonant wavelength, injecting a '0' value. By a procedure consisting of introducing a bias voltage to the microring, its resonant wavelength is shortened, which makes the microring be out of resonance. This prevents the destruction of light, resulting in the injection of a '1' value. The desired stream of '0's and '1's for the transmission is modulated by bringing the ring in and out of resonance appropriately. This stream can be read by another microring resonator acting as a detector.

When acting as a detector, the microring is always on-resonance, which causes any light of its resonant wavelength passing through the adjacent waveguide to be coupled into the microring (notice that this detection procedure removes the signal from the waveguide). In order to translate the optical signal into the electronic domain, typically a Germanium doped photodiode, embedded in the microring, detects a logic '1' each time that light is coupled into the microring, and detects a logic '0' when no light is coupled into the microring. Very high operating frequencies have been demonstrated for this modulation and detection procedure (e.g., 10 GHz), providing high bitrates. Each of the channels of a waveguide on which WDM is used requires dedicated modulator and detector microrings with the right optical path length to be on-resonance with the wavelength of the channel. Note that microring resonators enable multiple data channels on a single waveguide easily by using different non-colliding wavelengths, increasing bandwidth density. In addition, microrings are very effective in terms of modulation energy per bit.

Figure 2.4 shows all the elements involved in the process of transmitting data with photonics. A light source injects all the wavelengths that carry data (two in this case) into the waveguide. Specific microrings exist to modulate and to detect information encoded in each wavelength. The microring resonators injecting data *destroy* the light corresponding to zeros and let pass the light corresponding to ones in the data stream. The microring resonators reading the data destroy the remaining light, which is interpreted as '1's, and also detect the lack of light, which is interpreted as '0's.

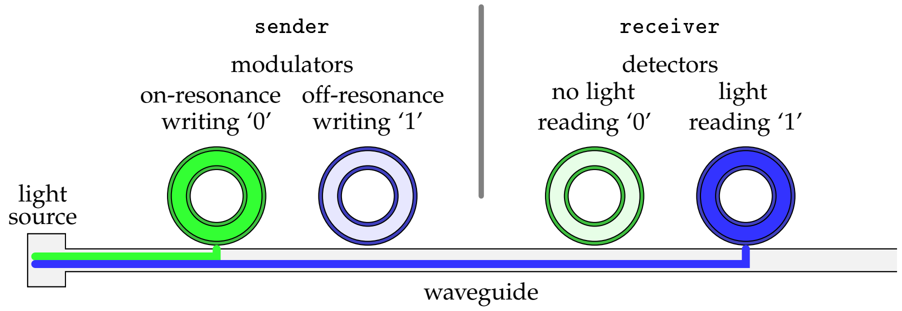


Figure 2.4: Basics of photonic data transmission with just two wavelengths (blue and green). Each microring resonator is associated with one wavelength that makes the microring be on resonance. Four microring resonators appear in the figure, a modulator and a detector for each wavelength. A modulator on resonance *destroys* the light of its associated wavelength, causing the corresponding detector to read a '0' value (see the green wavelength). A modulator electrically made be off-resonance lets light pass, and the corresponding detector reads a '1' value as a result (see the blue wavelength).

Chapter 3

Multiple-Area Cache Coherence Protocols

Server consolidation consists of running multiple services on a single physical server to increase its utilization, reducing the operating costs, the number of machines needed and the energy consumed to run these services. Server consolidation is typically implemented by running services in virtual machines.

With power consumption as the major constraint when scaling up the number of cores in a chip, server consolidation is rising in parallel as an appropriate instrument to take advantage of such large numbers of cores. In this chapter, we describe mechanisms to reduce the power consumption of chip multiprocessors used for consolidated workloads by means of cache coherence protocols whose operation is designed to fit this scenario. For this, we statically divide the chip in areas from the point of view of the coherence protocol, while keeping a single coherence domain for simplicity and compatibility. This allows us to increase data locality without incurring higher LLC miss rates, and to reduce the directory overhead needed to support cache coherence. This translates into less power consumption without performance degradation. Moreover, these mechanisms can also provide noticeable performance gains in some cases thanks to increased data proximity. Cache coherence is maintained per area and pointers are used to link the areas, thereby achieving isolation among virtual machines and savings in storage requirements.

Additionally, the coherence protocol dynamically selects one node per area as the one responsible for providing the data on a cache miss, thus reducing the number of retransmission points, lowering the average latency of cache misses and the traffic circulating among areas. Compared to a highly-optimized directory implementation, the leakage power consumption of the directory is reduced by 54% and the dynamic power consumption of the caches and the network-on-chip is reduced by up to 38% for a 64-tile chip multiprocessor with 4 virtual machines, showing no performance degradation. Moreover, some noticeable speed-ups are attained, such as a 6% performance improvement in the well-known apache web server. In general, our proposals result in improved scalability of cache coherence for higher core counts, with increasing benefits in both terms of storage requirements and data locality.

3.1 Background

In the current energy constrained scenario, the so-called power wall is the main concern in the development of CMPs, as it may prevent these chips from integrating the expected amount of cores due to associated excessive power consumption. To tear down this wall, the design of every element of the chip should aim at reducing power consumption.

In a shared-memory architecture like the tiled-CMPs assumed in this thesis, the cache coherence protocol is key in the performance and power consumption of the whole system. For instance, NoCs and cache memories have been reported to account for up to 50% of the overall power consumption of the chip [121], showing an ever-increasing trend. Cache coherence protocols are in charge of orchestrating and making use of these elements, with a great influence in their final contribution to power consumption. At the same time, the storage requirements of cache coherence may limit its efficient applicability to large core counts. This apparent lack of scalability of NoCs and cache coherence mechanisms will limit CMP evolution unless computer architects provide mechanisms with better scalability.

Under these circumstances, when the system is meant to be used mainly in a particular environment, the coherence protocol should take advantage of its special characteristics in order to improve performance and reduce power consumption, without losing sight of the need to also optimize general uses of the chip. Recently, virtualization has received a lot of attention with the popularization of cloud computing [124]. As the number of cores in chip multiprocessors (CMPs) increases and limited parallelism in current applications makes it difficult to take advantage of large core counts, virtualization is gaining importance to use multiple cores by running many isolated virtual machines (VMs) in a single chip. However, there is a lot of research to do in the design of chips adapted to virtualization, especially when taking into account the necessity to reduce the power consumption of these chips if they are to integrate hundreds of cores soon.

Virtualization poses many opportunities to chip designers due to its special characteristics. Virtual machines run in almost complete isolation, a circumstance that can be taken advantage of by the cache coherence protocol to improve the operation of the system by providing higher data locality and smaller area footprint for cache coherence information. This would help prevent NoC and cache coherence scalability from becoming a serious issue, which is the goal of the mechanisms described in this chapter. However, there are several caveats to avoid when doing so.

To understand the basics of the operation of our proposals, first we have to analyze the data types present in virtualized workloads. Typically, there are two kinds of data in a VM: private data to the VM and data shared between VMs. Private data is only accessed by a single VM (but possibly by several threads of the same VM), and therefore, there is no need to keep coherence information beyond the limits of that VM.

On the other hand, almost all the data shared between VMs is expected to be read-only data to which the hypervisor has applied memory deduplication [93, 174]. Deduplicated memory pages are read-only memory pages, with identical contents, that are present in the virtual memory of more than one single VM. The hypervisor usually detects these identical pages in different VMs, at run time, and allocates a single physical page in physical memory for all of the VMs to share. If a deduplicated memory page is written by a VM, a copy-on-write policy ensures that a new physical page is allocated to be used by the writer of the page, which is removed from the group of sharers of the deduplicated page. Memory deduplication can provide large memory savings, which has made this technique very popular for server consolidation. Linux has supported memory deduplication in the KVM hypervisor since version 2.6.32, and other hypervisors such as Xen [93] or VMware [174] already support it.

Regarding the cache coherence protocol, if cache coherence were kept strictly per VM, significant savings in coherence information storage could be achieved in cache. However, doing this naively implies that deduplicated data would need to be reduplicated at the last-level cache to give each VM its own locally-tracked copy, which would increase cache pressure and thus reduce performance. Previous research shows that, in the presence of deduplication, system performance improves by 6.6% on average if the pressure on the last-level cache is reduced by storing in it a single copy of deduplicated data [61], assuming a scenario with just moderate deduplication.

As an example of the importance of deduplication, note scenarios such as Amazon Elastic Compute Cloud (Amazon EC2) [170], in which a multitude of virtual machines run in isolation from one another. Assuming servers containing CMPs with multiple cores, each core typically runs one server instance. The same operating systems and libraries will be widely used among many of these instances, as well as some common applications such as web servers. Deduplication can notably reduce the amount of main memory required, used by read-only data as well as operating system and application code, by means of storing a single copy of each of these shared pages in memory, instead of one copy per VM. Such enormous storage benefits could also be transferred to on-chip cache level, boosting performance, if the coherence protocol were designed to do so.

The novel coherence schemes presented in this chapter address some of the opportunities posed by server consolidation: they reduce power consumption, reduce the area overhead of cache coherence, keep a single copy of deduplicated data in the shared cache, provide (partial) isolation among cores of different VMs and reduce the average latency of cache misses by increasing data locality. To the best of our knowledge, the cache coherence protocols described in this chapter are the first ones presenting such properties.

The basic ideas behind our proposals —which will be elaborated later— are introduced at this point to put the remaining of this background section in perspective. First, our proposals try to isolate each VM in a different area of the chip (an area is a subset of all the tiles of the chip). This is achieved by instructing the OS to schedule the threads of each VM to tiles in different areas, although any configuration of VMs is possible with a small cost in performance when a VM uses tiles from more than one area. Equivalently, applications running on several areas —even if just one application uses all areas— still benefit from the features of these protocols, making them flexible for any situation. By doing this, apart from avoiding interference between cache accesses of different virtual machines, coherence information can be kept per area, which significantly reduces the storage overhead of the proposed coherence protocols and therefore their power consumption.

Second, our proposals are based upon Direct Coherence (DiCo) [155], which will be described in Section 3.1.2, and whose characteristics make it a suitable baseline for the consolidated scenario in a CMP, especially due to its ability to resolve cache misses in just two hops without visiting the home node for the memory block. Our proposals leverage DiCo's characteristics to resolve misses inside the area, increasing data locality.

We have derived two cache coherence protocols from DiCo. One of them, called DiCo-Providers (Section 3.2.1), is also well suited for non-virtualized environments while the other, named DiCo-Arin (Section 3.2.2), trades some performance in the general case for increased simplicity and reduced power consumption in virtualized scenarios. In order to allow the presence of sharers for a block in any L1 cache of the chip, one L1 cache in each area (the *provider*) tracks the sharers in its area in DiCo-Providers, while DiCo-Arin does not keep exact information about sharers from more than one area and relies on broadcast to invalidate them when necessary.

3.1.1 Base Architecture

We assume the tiled-CMP organization described in Section 1.4, using an optimized directory-based coherence protocol as our baseline in this chapter. The L2 cache, although physically distributed, is organized in a last-level cache which is logically shared among all tiles. For each memory block, a subset of bits of the address —the "bank label"— determines the home L2 bank (the bank that caches the block and the coherence information for that memory block). To store the directory information for those blocks not present in the L2 cache, we use the same approach as NCID [187], in which extra tags in the L2 cache are used to store a virtual directory cache. This results in L1 and L2 caches that are non-inclusive, increasing cache capacity while avoiding the need to perform snoopy for any requests. Additionally, if a block is evicted from the L2 cache, the directory information remains in the NCID directory cache, preventing invalidations of the block in the L1 caches. Only when a directory entry is evicted, the block is also evicted (if present), and every copy of the block is invalidated. When copies of the block are present in the chip, the home L2 for the block stores their directory information. Upon an L1 cache miss, a request is sent to the home L2 bank, where the directory information for the block can be found, to carry out the steps required to resolve the cache miss.

It is important to note that we use full-map bit-vectors instead of coarse bit vectors [69], limited pointers [4, 32] or any other sharing codes because full-map bit-vectors provide the best performance and lowest traffic for the base architecture. Other sharing codes trade off reduced directory overhead for extra network traffic and worse performance, while our proposals improve the results in all of these metrics. Nevertheless, our protocols are orthogonal to the particular sharing code used, and they could be implemented in combination with alternative sharing codes to further reduce the directory overhead if desired.

3.1.2 Direct Coherence

Our protocols are based upon a Direct Coherence (DiCo) scheme [155]. In DiCo, the coherence information and the ownership of the block are stored along with the data in the L1 caches. This makes it possible to resolve most misses in just two hops (i.e., without directory indirection) by predicting the destination of requests upon L1 cache misses. When the prediction is correct, requests are sent straight to the owner L1 cache, which answers with the data and takes note of new sharers. Additionally, upon a write miss in the owner L1 cache, it can send invalidations to the sharers because it knows who they are. In this way, the distinctive indirection of directory protocols is avoided for most misses.

As for deduplicated data, direct coherence does not force its reduplication in the shared L2 cache. Only one copy of the data is needed for all the tiles, reducing the space needed for shared data in cache compared to other proposals such as Virtual Hierarchies [128] or Coherency Domains [23,101,140] (Section 3.5).

We use DiCo as a baseline because of its ability to resolve most misses in two hops without visiting the home node (just the owner node) because its prediction technique has been proven very accurate [155]. Since owner L1 caches are located within the tiles running the VM, while the home L2 cache can be located anywhere in the system, DiCo is a very suitable protocol for the environment considered in this work.

3.2 Multiple-Area Coherence Protocols

DiCo provides some desirable features when used in a consolidated server. DiCo isolates the VMs running in the server and brings data closer to the requestors —data is found within the VM—, increasing data locality without incurring higher LLC pressure with private copies of blocks. DiCo also reduces the number of hops upon a cache miss (two hops in DiCo instead of three as in an ordinary directory-based protocol). All this is possible because, in the common case, the directory information for a block private to a VM can be found in an L1 cache belonging to that VM. This data can be accessed in just two hops, without needing to send any coherence messages out of the VM.

In addition, with DiCo, deduplicated data is not reduplicated, with beneficial effects on performance, as discussed earlier. The owner node and associated directory information of a block is present in only one of the VMs. Unfortunately, this behavior also implies that when other VMs need to access the block (e.g., deduplicated data), they must send their read requests to the owner, in a different VM, in order to get the block. This results in higher latency and power consumption for these misses with respect to misses resolved within VMs (e.g., data private to the VM), also creating traffic interferences among VMs.

To turn this situation into an advantage, we propose to statically divide the CMP in a fixed number of areas (subsets of adjacent tiles of the chip), on top of DiCo. The use of a static division of the chip in areas enables a reduction of the directory information, and therefore of its power consumption, contrary to a dynamic division of the chip that would increase the directory size to support all the possible configurations of the chip, increasing the power consumption of the caches.

Our claim is that we can take advantage of the circumstance that data shared across VMs is expected to be read-only data in most cases. Based on that, our novel coherence scheme allows L1 cache load misses for such data to be resolved inside the area, with less power consumption, while keeping a single copy of deduplicated data in the shared level of the cache, boosting performance on both fronts. And at the same time, directory information gets noticeably reduced.

The division of the chip in areas is hard-wired. The OS or hypervisor should be made aware of the different areas in the chip to better map processes to cores and provide isolation between VMs and better performance, although the system would work correctly anyway. The hardware needs no information about the VMs running on top of it.

The hard-wired division in areas is not an important issue when the VMs do not exactly match the areas, as the experimental results of the evaluation section prove. For instance, when an application uses all the cores of the chip, the data shared by several areas can still be accessed without leaving the areas of the requestors, so we still have the benefits of increased data locality. In addition, we have the power benefits of the smaller directory entries enabled by the static division of the chip. This makes our proposals attractive for scenarios other than consolidated servers using virtualization.

Next, we describe two protocols that, following the principles just described, reduce the latency and power consumption of misses to deduplicated data while keeping a single copy in L2 cache.

3.2.1 DiCo-Providers

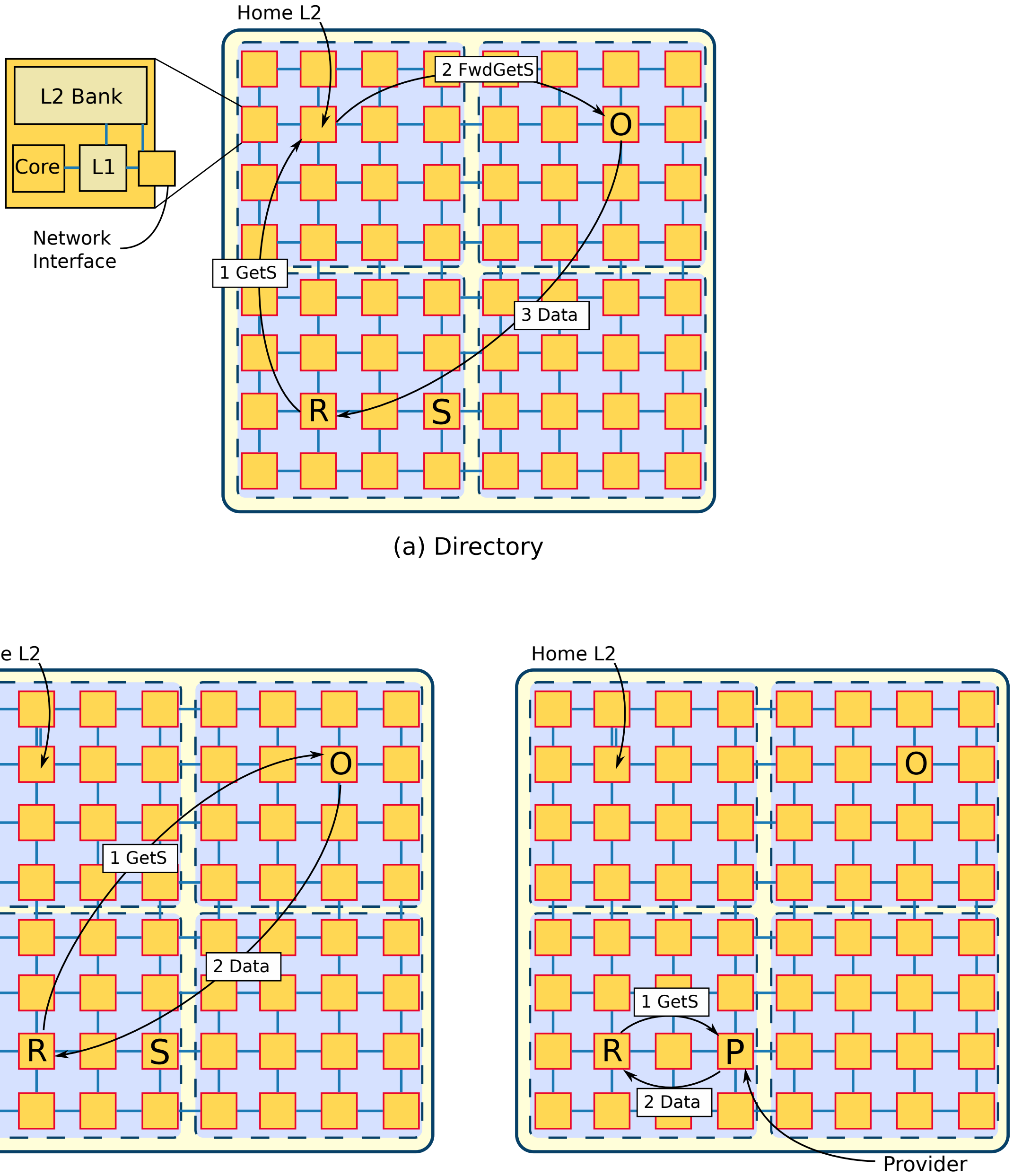
In DiCo-Providers, coherence information is kept for the sharers of an area, instead of for the whole chip, thereby reducing the storage requirements of sharing information. If no VM uses tiles of more than one area, only deduplicated data will be shared between areas, requiring inter-area tracking. The rest of the data will just require intra-area tracking.

In DiCo-Providers, we introduce the *providership* concept and its associated state, the *provider* state. A core storing a block in provider state in its private cache can directly reply to read requests from other cores of its area. Every area can have its own provider for every block shared between areas, hence allowing the resolution of requests to these data in two hops without leaving the area. However, a single ordering point remains in the chip, the owner, which is located in one of the areas and ties together the coherence information of all the areas sharing the data. To simplify, the term *supplier* is used to refer to a node that can be either an owner or a provider.

To illustrate the differences between protocols, the common operation to resolve an L1 miss for a deduplicated block can be seen in Figure 3.1 for the directory protocol, DiCo and DiCo-Providers.

In DiCo-Providers, the directory information of each data block is distributed across the chip. Just like in the base architecture, every block has a fixed home L2 bank, which is determined by using several fixed bits of its address. The ownership of a block can be held by its home L2 bank or by any L1 cache. If the ownership is held by an L1 cache, then the home L2 bank keeps a pointer in a special DiCo structure (called L2C$, see Section 3.3) to store the current location of the ownership. The owner (be it an L1 cache or the home L2 bank) keeps the directory information about the providers (up to one provider per area). The directory information regarding the sharers of each area is kept by the local provider of the area. When an L1 cache holds the ownership, it also keeps the coherence information about the sharers in its area. That is, the owner L1 cache behaves as the only provider for its local area. Notice that if the home L2 bank holds the ownership, it does not keep coherence information about sharers, as that information is stored by the providers. Figure 3.2 compares the distribution of coherence information in a flat-directory, the original DiCo protocol and DiCo-Providers.

In DiCo-Providers, there are three events initiated by a core that cause its L1 cache to become the owner of a block: (1) a write request; (2) a read request when the block is not present in the chip; (3) a read request when there is no provider

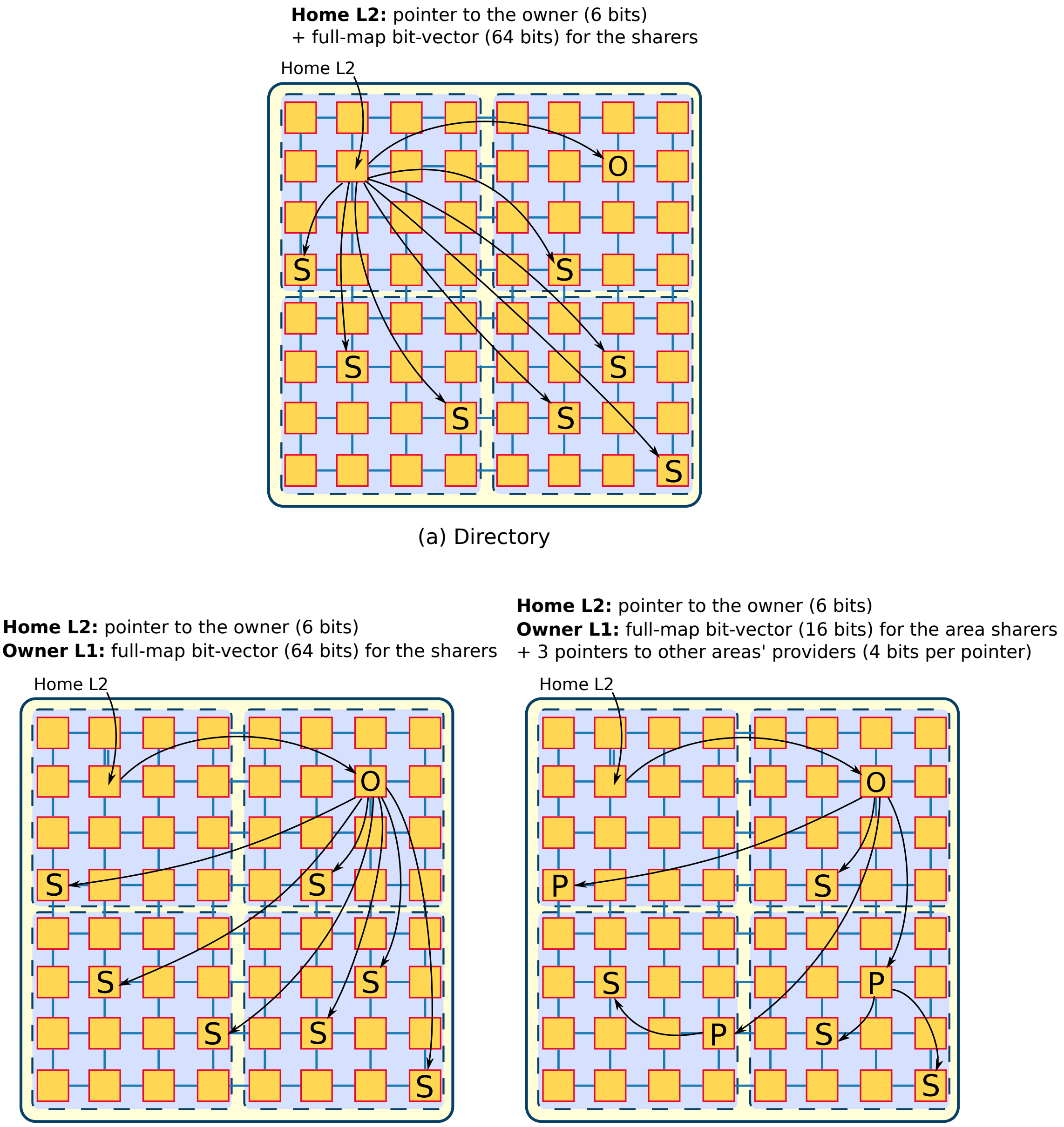


(a) Directory

(b) Direct Coherence

(c) DiCo-Providers

Figure 3.1: Read request to a deduplicated block. Four VMs running on the chip (dashed lines). One sharer exists in the requestor's area. (a) Directory indirection causes a long 3-hop miss. (b) DiCo avoids one hop by sending the request straight to the owner in L1 cache. (c) DiCo-Providers additionally reduces the number of traversed links by sending the request to the sharer (Provider) in the area. (O = Owner; S = Sharer; P = Provider; R = Requestor).



Home L2: pointer to the owner (6 bits)
+ full-map bit-vector (64 bits) for the sharers

(a) Directory

Home L2: pointer to the owner (6 bits)
Owner L1: full-map bit-vector (64 bits) for the sharers

Home L2: pointer to the owner (6 bits)
Owner L1: full-map bit-vector (16 bits) for the area sharers
+ 3 pointers to other areas' providers (4 bits per pointer)

(b) Direct Coherence

(c) DiCo-Providers

Figure 3.2: Coherence information (valid bits not shown). DiCo-Providers needs noticeably less storage for coherence information than the directory and DiCo. Four areas assumed. (O = Owner; S = Sharer; P = Provider).

in the area and the block ownership is held by the home L2 bank. Another event initiated by a different L1 cache, the replacement of the block by the owner L1 cache, also causes an ownership transference between L1 caches, as discussed in Section 3.3.1.1.

In turn, the home L2 bank becomes the owner of the block only when the replacement of the ownership by an L1 cache takes place in an area with no remaining sharers, as discussed in Section 3.3.1.1.

As for providership, only L1 caches can be providers, not L2 banks. There are two ways for an L1 cache to become the provider of a block: (1) the core using that L1 cache performs a read request in an area with no supplier while an L1 cache in another area already holds the ownership; (2) because of a providership transference due to a block replacement (Section 3.3.1.1).

3.2.2 DiCo-Arin

Unfortunately, DiCo-Providers' operation shows great complexity if compared to the original DiCo and other directory-based protocols. The providership and ownership transferences present in DiCo-Providers cause a number of coherence protocol races that complicate its correct implementation.

For this reason, we have polished and simplified our general proposal by optimizing it for the virtualized scenario at the cost of some performance in the general case, taking advantage of the fact that deduplicated pages are expected to be read-only pages and that, in turn, the data shared between the areas in the chip is expected to be deduplicated data. The result is DiCo-Arin: a protocol with similar complexity to the original DiCo in which no precise information about sharers is kept for data shared between areas. Instead, a simple broadcast mechanism is used to invalidate all the copies of these blocks when needed, which should be very infrequently due to the expected read-only nature of that data. Our broadcast mechanism is never used to locate data to answer a read request, since at least one copy of the data can always be found by using the available directory information. We make sure that our broadcast approach is safe by adding some constraints, as discussed in Section 3.3.2.1.

DiCo-Arin also tackles a potential problem in the operation of DiCo-Providers. In DiCo-Providers, the critical path of a few misses is five hops. In particular, this happens when a read miss for a block shared between areas takes place, originating from an area containing a provider, and with the misfortune that the resulting request is sent to a mispredicted L1 cache (first hop). The mispredicted L1 cache forwards the request to the home L2 bank (second hop). Then, the home

L2 bank forwards the request to the L1 cache that holds the ownership (third hop), who forwards it to the provider in the requester's local area (fourth hop) who finally responds with the requested data (fifth and final hop) and updates the directory information of the block (by adding the requestor). Fortunately, this situation does not happen frequently, but it could become an issue under pathological cases.

To avoid such kind of misses, we decided for DiCo-Arin that a copy of any block shared between areas will be always stored in the home L2 bank, so that requests to such blocks will be answered directly by the home L2 bank, avoiding two of the hops incurred by DiCo-Providers, corresponding to reaching the owner L1 cache and the local area provider.

Once we assume the previous optimization, two different ways to keep coherence for blocks shared between areas are possible. The first one consists of keeping precise directory information, which would require some mechanisms to coalesce all the directory information as new sharers appear, including those sharers that received their data straight from the home L2 bank instead of from the owner L1 cache. This, however, would increase the complexity of the protocol and, as previously stated, one of the main purposes of DiCo-Arin is to provide a simpler alternative to DiCo-Providers. For that reason, we chose the second way, which consists of keeping inexact directory information for data shared between areas and then relying on a broadcast mechanism to find all the sharers of the block when needed, which should be infrequently due to the read-only nature expected from the data shared between areas.

The design of DiCo-Arin is such that the protocol behaves the same as the original DiCo protocol for blocks which have all their copies confined to one area of the chip. However, as soon as a read request coming from a remote area reaches the owner L1 cache, the data and its ownership is transferred to the home L2 bank and its former holder becomes a provider for the block. The home L2 bank will then be able to serve the block to read requests as any other supplier. Notice that in the case that the home L2 bank was already the owner —due to a previous ownership replacement from an L1 cache—, the last step —sending the data to the home L2 bank— is not necessary. These blocks shared between areas have no precise directory information stored anywhere and they rely on broadcast for invalidation (the ordering point is the home L2 bank in that case). In addition, every new sharer of the block can also act as a provider, giving rise to opportunities for further protocol optimization.

3.3 Detailed Operation of the Protocols

In order to better understand the proposed protocols, it is first necessary to introduce a few concepts.

Regarding the areas of the chip, we must differentiate the *local area* from the *remote areas*. When talking about an L1 cache, its local area is the area that the L1 cache belongs to. Any other area is a remote area for the L1 cache.

We use two kinds of pointers that hold sharing information in our protocols to point to L1 caches: the general pointer type, named *GenPo*, and the type of pointers to providers, named *ProPo*. The size of a GenPo is $\log(ntc)$, where ntc is the number of tiles in the chip. Thereby, a GenPo can point to any L1 cache of the chip. The size of a ProPo is $\log(nta)$, where nta is the number of tiles in each area. Hence, given one area, a ProPo can point to any tile in that area. Notice that a GenPo is larger than a ProPo. Since a GenPo can point to any tile, it can be used to point to an owner or to a provider, while a ProPo can only be used to point to a provider, given that the area to which the provider belongs is known.

Two structures of the DiCo protocol that the reader might not be familiar with are the *L1 Coherence Cache* (L1C$) and the *L2 Coherence Cache* (L2C$). The L1C$ is indexed by block address, with each entry containing an address tag and a GenPo. The GenPo holds the identity of a possible supplier for the block —the owner or a provider L1 cache—, to be used as a prediction upon the next cache miss to the block. Upon an L1 miss, this prediction —if available— is used as the destination for the request. Otherwise, the request is sent to the home L2 bank of the block. The mechanism to update the L1C$ is detailed in Section 3.3.1.2. In general, when a block is evicted from the L1 cache, the identity of its supplier is retained in the L1C$ to resolve future cache misses in two hops. The reuse of blocks provides the L1C$ with a good hit ratio [155].

As for the L2C$, it is a cache co-located with the L2 banks, indexed by block address that, similarly to the L1C$, contains address tags and GenPos. Contrary to the L1C$, the information in the L2C$ is not a prediction but the precise identity of the L1 cache that holds the ownership for the block.

3.3.1 DiCo-Providers

First, we discuss the operation of the protocol upon an L1 miss. The process starts by checking the L1C$ for a supplier prediction. If a hit occurs in the L1C$, the request is sent to the predicted L1 cache. Otherwise, the request is sent to the home L2 bank. The objective of using the L1C$ is to resolve the request without

Table 3.1: Actions performed in DiCo-Providers upon the reception of a request (excluding the updating of L1C$ and L2C$).

Request Type	Receiver cache	State	Request coming from local area	Provider Exists	Owner in L1 cache	Actions taken
read	L1	owner	yes			Send data. Store coherence info in bit vector (requestor becomes sharer)
			no	yes		Forward request to provider in remote area
				no		Send data. Store coherence info in ProPo (requestor becomes provider)
		provider	yes			Send data. Store coherence info in bit vector (requestor becomes sharer)
			no			Forward request to home L2 bank
		other				Forward request to home L2 bank
	L2	owner		yes		Forward request to provider
				no		Send data. Store coherence info in the L2C$ (requestor becomes owner)
		other			yes	Forward request to owner
					no	Send request to memory controller to fetch data from memory. Store coherence info in the L2C$ (requestor will become owner in exclusive state)
write	L1	owner				Start invalidation. Send data. Send Change_Owner message to home L2 to store coherence info in the L2C$ (requestor becomes owner in modified state)
		other				Forward request to home L2 bank
	L2	owner				Start invalidation. Send data. Store coherence info in the L2C$ (requestor becomes owner in modified state)
		other			yes	Forward request to owner
					no	Send request to memory controller to fetch data from memory. Store coherence info in the L2C$ (requestor will become owner in modified state)

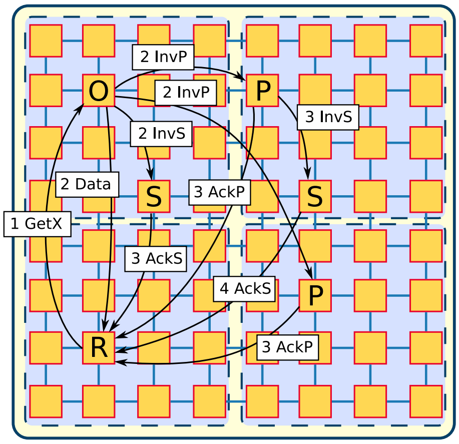


Figure 3.3: Write request and invalidation process. The supplier prediction succeeds.

indirection (two hops instead of three) by sending it straight to the supplier in the local area. If a *misprediction* occurs, the request reaches an L1 cache that cannot provide the data, which forwards the request to the home L2 bank to determine there the following steps to take based on the available coherence information.

Table 3.1 describes the actions performed by caches upon the reception of a request. The request is forwarded as many times as necessary until it reaches a supplier. The original deadlock-avoidance mechanism of DiCo applies to our protocol to prevent a message from being forwarded indefinitely [155].

The invalidation process on a write miss can be seen in Figure 3.3. The owner (ordering point) invalidates the sharers in its area and the providers in other areas, which in turn invalidate the sharers in their areas. The owner also sends a message, which we call Change_Owner, to the home L2 bank (not shown in the image) to let it know the identity of the new owner (the requestor). The ownership cannot be transferred again until an acknowledgement from the home L2 bank, in response to the Change_Owner message, is received by the new owner. This prevents a stale owner from being stored in the L2C$ of the home L2 bank due to the unordered arrival of consecutive Change_Owner messages for the same block, as this mechanism ensures that at most one Change_Owner message concerning each block is being processed at a given time.

Two counters are needed in the miss status holding register (MSHR) of the requestor: one counter to track the number of pending acknowledgement messages from the providers, and a second counter to track the number of pending acknowledgement messages from the sharers. The latter counter is

Table 3.2: Actions taken by DiCo-Providers upon a block replacement from L1.

Block state	Sharers exist in the area	Actions taken
shared		Silent eviction
provider	yes	Send providership and sharing code to a sharer (the sharer will send a Change_Provider message to the owner)
	no	Send No_Provider to the owner
owner (including any exclusive state)	yes	Send ownership and sharing code to a sharer (the sharer will send a Change_Owner message to the home L2)
	no	Send ownership (and data if dirty) to the home L2

incremented every time an acknowledgement message from a provider is received, containing the number of sharers in its area (from which acknowledgement messages will also be received). The invalidation process is complete when both counters reach zero. We need separate counters to prevent protocol races while enabling the concurrent invalidation of all the copies of the block.

One special case is that in which the requestor of a write request is also a provider. When this happens, the requestor is in charge of invalidating the sharers in its area. However, the invalidations cannot be sent until the requestor receives the ownership or an invalidation message. The latter case happens when a write request from another L1 cache is being served before the request issued by the provider.

Upon the eviction of a block from its home L2 bank, every sharer in the chip must also be invalidated, and the mechanism employed for this is basically the same as the one just explained to resolve a write request. In this case, the L2 cache acts as both the owner (by sending the invalidations) and the requestor (by receiving the acknowledgements).

3.3.1.1 Block and L2C$ Information Replacements

Table 3.2 describes the actions carried out to replace L1 cache blocks depending on the state of the block to be evicted. Replacements may involve ownership and providership transferences.

As in write requests, each Change_Owner message is followed by the reception of an acknowledgement message from the home L2 bank before the ownership can be transferred again, in order to prevent races. The same applies to Change_Provider messages.

Sometimes, due to a previous silent L1 replacement, the cache chosen to receive the ownership or providership cannot accept it. In this case, the ownership is forwarded to another sharer. If no sharers exist, the ownership is transferred to the home L2 bank.

Another type of replacements are those involving L2C$ information. The L2C$ has a limited number GenPos, and they may need to be evicted upon conflicts. When this happens, a message is sent to the owner to make it relinquish the ownership and send it back to the home L2 bank along with the identity of the providers and the data (if dirty). When the ownership is transferred to the home L2 bank, the former owner L1 cache becomes the provider for its area.

3.3.1.2 L1C$ Update Mechanism

Upon L1 cache misses, the prediction of the destination of requests is based on the address-indexed contents of the L1C$, which stores pointers to L1 caches that are potential suppliers of the block. The information stored in the L1C$ should be as precise as possible in order to achieve a high ratio of correct predictions, although it is to note that incorrect information affects only the performance of the system, not its correctness.

In addition, L1 cache entries can store potential suppliers at no additional cost with respect to DiCo, as a GenPo can be easily accommodated in the space used by DiCo to store directory information when the L1 cache holds the ownership. This way, cached blocks do not take up pointers in the dedicated array of the L1C$. The pointers in L1 cache entries are considered part of the L1C$ and looked up too when making a prediction.

Figure 3.4 shows the three possible states for the prediction of a block: in the L1 cache, in the L1C$ or not present. Since the objective is to store the identity of a potential supplier, those messages sent by a possible supplier (data messages, invalidations and write requests) are used to update the predictions of the block.

Also, some hint messages, whose only purpose is to update the prediction information, are sent as part of the regular operation of DiCo, for instance to let the sharers know the identity of the new owner or provider when the ownership or providership moves.

3.3.2 DiCo-Arin

Like DiCo-Providers, DiCo-Arin is a provider based protocol, which means that a number of nodes, called providers, can serve data to read requests in addition to

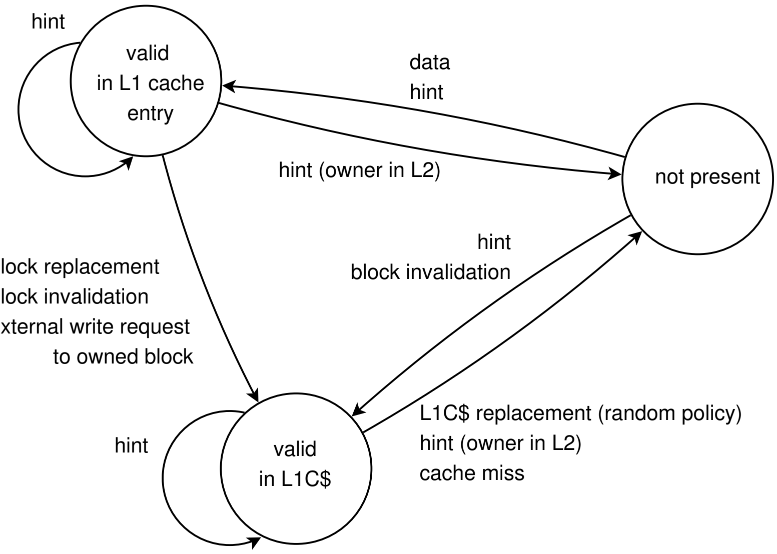


Figure 3.4: State diagram for the prediction of the supplier of a block.

the owner. However, contrary to DiCo-Providers, where the directory information regarding the providers was located in the node holding the ownership, in DiCo-Arin this information (up to one provider per area) is always located in the home L2 bank along with a copy of the data (which is necessarily shared between areas). This way, when a read request to data shared between areas reaches the home L2 bank in DiCo-Arin, the information about the provider (if present) is sent along with the data to the requestor, so that the requestor can store the identity of its local-area provider in its L1C$. These requestors may solve future misses within their areas in two hops by using the provider information. If there was no provider in the area, the home L2 bank stores the requestor as the provider for the area for future reference.

Due to the simplicity of DiCo-Arin, provider information might become stale in the L2 cache, as no careful management of providership is performed, and no explicit Change_Provider messages are required. In order to keep provider information updated, when a request to data shared between areas is forwarded by a mispredicted L1 cache and reaches the home L2 bank, the protocol checks whether the provider stored in the home L2 bank for the area matches the identity of the L1 cache that forwarded the request. If they are the same, it means that the provider information is stale. In that case, the new requestor is stored in the home L2 bank as the new provider for the area in future references. To enable the comparison, the identity of the forwarder of a request is included in the forwarded message.

We have implemented an optimization consisting of turning L1 caches into providers, instead of sharers, when they request a copy of a block shared between areas. Thanks to this, read requests are more likely to find a provider based on L1C$ information, even if this information is stale. Notice that this was not possible when exact information was kept for blocks. Sharers were not allowed to provide the data, as it would result in new sharers not being tracked by the owner or the provider. This is not a problem in DiCo-Arin, which does not need to keep directory information about the copies of blocks shared between areas. We can allow sharers to act as providers for such data, and new sharers will be located by the broadcast mechanisms of DiCo-Arin when necessary.

3.3.2.1 Ensuring Safe Broadcast Invalidations

As we explained in Section 3.2.2, DiCo-Arin uses a broadcast mechanism to invalidate the copies of a block shared between areas upon the occurrence of a write request or the eviction of a block from its home L2 bank.

In order to use a broadcast mechanism, we must ensure that it does not break the correctness of the protocol. Broadcasts should never interfere with other requests for the block causing unexpected results (like deadlocks), and coherence should not be violated for the block as a result of broadcasts (i.e., no copies of the block must remain in the chip after its invalidation).

To ensure these conditions, we use a three-way invalidation mechanism. First, the home L2 bank of the block broadcasts the invalidation message. When this message is received, the L1 caches lock the block, in order not to respond to requests regarding the same block during the invalidation process. Second, every L1 cache acknowledges the invalidation to the requestor or the home L2 bank, depending on whether the invalidation was caused by a write request or an L2 cache eviction, respectively. Finally, the receiver of the acknowledgements broadcasts another message to let the L1 caches unlock the block and issue responses to requests regarding that block again. The combination of three steps and block locking prevents race conditions that could result in incoherency. For instance, already invalidated L1 caches could obtain (incoherent) copies of the block from yet-to-be-invalidated L1 caches by means of prediction in the middle of the invalidation process. These race conditions are rare and involve the reception of stale hint messages, prediction success, and a very particular timing of all these messages, including the invalidation and acknowledge messages. Nevertheless,

Table 3.3: System configuration.

Processors	64 UltraSPARC-III+ 3 GHz. 2-ways, in-order.
L1 Cache	Split I&D. Size: 128KB. Associativity: 4-ways. 64 bytes/block. Access latency: 1 (tag) + 2 (data) cycles.
L2 Cache	Size: 1MB each bank. 64MB total. Associativity: 8-ways. 64 bytes/block. Access latency: 2 (tag) + 3 (data) cycles.
RAM	4 GB DRAM. 8 memory controllers along the borders of the chip. Memory latency 300 cycles + on-chip delay. Page Size: 4 KB.
Interconnection	Bidimensional mesh 8x8. 16 byte links. Latency: 2 cycles/link + 2 cycles/switch + 1 cycle/router (in absence of contention) Flit Size: 16 bytes. Control packet size: 1 flit. Data packet size: 5 flits.

the prevention of these races justifies the use of a three-step invalidation process, in order to maintain the correctness of the protocol, with a minimum impact in performance.

3.4 Evaluation

3.4.1 Methodology

We use Virtual-GEMS [56] to simulate a server running a number of consolidated workloads. To model the network, we use a version of Garnet [5] to which we have added broadcast support [44]. Memory access latency is modeled as a fixed number of cycles plus a small random delay. We have also performed simulations with a more detailed DDR memory controller model, finding that this does not affect the results. We evaluated a 64-core tiled CMP by means of simulations with 4 VMs running in a single server. Each VM executes its own operating system (Solaris 10) and runs in 16 tiles. Memory deduplication is activated in every simulation.

As explained before, the chip is statically divided in four square areas of 16 tiles for DiCo-Providers and DiCo-Arin. In our default configuration, we assume that the OS or the Hypervisor have been instructed to schedule the threads in such a way that each VM executes in tiles from a different area, taking as much advantage as possible from our protocols. We also test an alternative configuration in which the threads of each VM have not been carefully scheduled,

Table 3.4: Benchmark configurations.

Workload	Description	Size	Simulation	Performance Metric	Memory saved by deduplication
apache4x16p	Web server with static contents	500 clients per VM, 10ms between requests	4 16-core apache VMs	No. of transactions in 500 million cycles	21.72%
jbb4x16p	Java server	1.5 warehouses per tile	4 16-core jbb VMs	No. of transactions in 500 million cycles	23.88%
radix4x16p	Sorting of integers	1M integers	4 16-core Radix VMs	Average execution time of all the VMs	24.18%
lu4x16p	Factorization of a dense matrix	512x512 matrix	4 16-core lu VMs	Average execution time of all the VMs	32.71%
volrend4x16p	Ray-casting rendering	Head	4 16-core volrend VMs	Average execution time of all the VMs	19.77%
tomcatv4x16p	Vectorized mesh generation	256	4 16-core tomcatv VMs	Average execution time of all the VMs	36.82%
mixed-com	Commercial benchmarks: apache, jbb	See the size of the corresponding benchmarks	2 16-core apache VMs and 2 16-core jbb VMs	Weighted no. of transactions in 500 million cycles	15.74%
mixed-sci	Scientific benchmarks: Radix, Lu, Volrend, Tomcatv	See the size of the corresponding benchmarks	4 16-core VM: Radix, Lu, Volrend and Tomcatv	Average execution time of all the VMs	15.21%

resulting in every VM using tiles from more than one area. Experiments indicated that this configuration fits our protocols worse than running a single parallel application in all cores. For this reason, we use this second configuration to represent the worst-case scenario for DiCo-Arin. This alternative configuration, which is shown in Figure 3.5, is denoted by the suffix "-alt". Tables 3.3 and 3.4 show the system configuration and benchmarks used, respectively. Table 3.4 also shows the average memory savings provided by memory deduplication in our benchmarks, which all the tested configurations translate into savings in LLC storage space.

We use CACTI 6.5 [133] to calculate the power consumption (static and dynamic) caused by cache structures (tags, data and directory), based on their sizes and frequency of access in the various protocols, assuming a 32 nm process. In order for our measurements to be accurate, we consider every event of the evaluated cache coherence protocols in the calculation of power consumption, including the complete sequence of steps taken in invalidations, block replacements or directory information updates, among others. As for the network, we calculate the power consumed by message routing and flit transmissions. For this, we use the model proposed by Barrow-Williams et al. [13] because of its

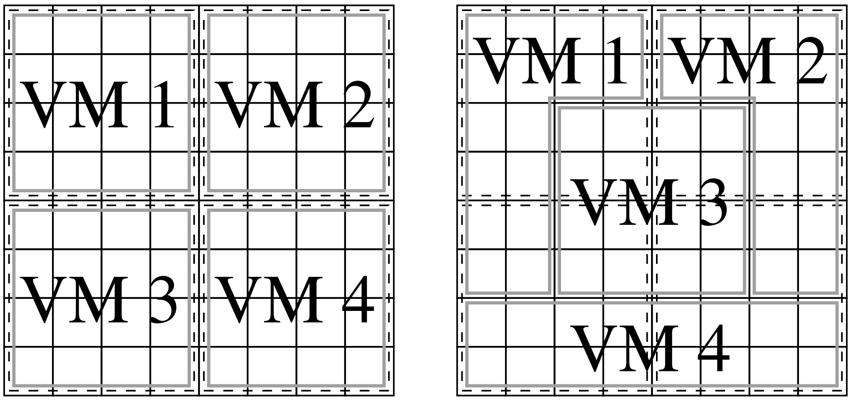


Figure 3.5: Configuration in which VMs fit the areas on the left. Alternative configuration on the right. Areas shown in dashed lines. VMs shown in grey lines.

simplicity. In this model, routing a message consumes as much energy as reading an L1 block, and four times as much energy as transmitting a flit.

3.4.2 Static Power Consumption

DiCo-Providers and DiCo-Arin provide significant savings in cache storage for directory information compared to DiCo and to a flat directory. This translates into static power savings.

To evaluate the static power consumption of the caches, first we have to introduce all the elements contributing to it. The evaluated 8×8 tiled CMP is divided in four areas. All the areas in the chip are composed by sixteen tiles. Table 3.3 shows the size of the caches. We assume physical addresses of 40 bits. There are five different types of tags in a tile: L1Tag (25 bits), L2Tag (17 bits), DirTag (17 bits), L1CTag (23 bits) and L2CTag (17 bits). GenPos have a size of 6 bits to point to any of the 64 tiles of the chip. ProPos have a size of 4 bits to point to any of the 16 tiles of an area. We also consider for some of the structures that the validity of an entry can be determined by the state of the block, while a separate valid bit is needed for some other structures.

Table 3.5 summarizes the amount of coherence information needed by each protocol. Contrary to the original DiCo, which needs even more coherence information than an ordinary directory protocol, DiCo-Providers reduces the overhead due to coherence information by 59% with respect to the flat directory

Table 3.5: Memory overhead introduced by coherence information (per tile) in our 8x8 tiled CMP. Four 16-core areas are assumed in DiCo-Providers and DiCo-Arin.

	Structure	Entry size	Entries	Total size (KB)	Overhead
Data	L1 cache	25 bits (L1Tag) + 512 bits (block)	2048	134.25	
	L2 cache	17 bits (L2Tag) + 512 bits (block)	16384	1058	
Directory	L2 dir. inf.	64 bits (sharers)	16384	128	12.56%
	Dir. cache	17 bits (DirTag) + 64 bits (sharers) + 6 bits (GenPo)	2048	21.75	
DiCo	L1 dir. inf.	64 bits (sharers)	2048	16	13.21%
	L2 dir. inf.	64 bits (sharers)	16384	128	
	L1C$	23 bits (L1CTag) + 6 bits (GenPo) + 1 Valid Bit	2048	7.5	
	L2C$	17 bits (L2CTag) + 6 bits (GenPo) + 1 Valid Bit	2048	6	
DiCo-Providers	L1 dir. inf.	16 bits (sharers) + 3×4 bits (3 ProPos) + 3 Valid Bits	2048	7.75	5.14%
	L2 dir. inf.	4×4 bits (4 ProPos) + 4 Valid Bits	16384	40	
	L1C$	23 bits (L1CTag) + 6 bits (GenPo) + 1 Valid Bit	2048	7.5	
	L2C$	17 bits (L2CTag) + 6 bits (GenPo) + 1 Valid Bit	2048	6	
DiCo-Arin	L1 dir. inf.	16 bits (sharers)	2048	4	4.49%
	L2 dir. inf.	16 bits (sharers) + 2 bits (area number)	16384	36	
	L1C$	23 bits (L1CTag) + 6 bits (GenPo) + 1 Valid Bit	2048	7.5	
	L2C$	17 bits (L2CTag) + 6 bits (GenPo) + 1 Valid Bit	2048	6	

and DiCo-Arin reduces it by 64%. Next, we describe the requirements of each protocol in detail to complement the information in Table 3.5.

In the case of the flat directory, each L2 cache entry contains a full-map bit-vector to track the sharers of the block, requiring one bit per core. In addition, a directory cache is needed to track the blocks in exclusive state in the L1 caches. Each entry of this directory cache contains a full-map bit-vector (to store the sharers), a GenPo (to store the identity of the owner L1 cache) and a DirTag.

In DiCo, a full-map bit-vector is needed in each entry of both the L1 cache and the L2 cache. The L2C$ requires one GenPo (for the case in which the ownership is held by an L1 cache) and an L2CTag. Each L1C$ entry stores one GenPo (for supplier prediction), and one L1CTag.

In DiCo-Providers, the only directory information that must be stored along with each block in the home L2 bank is one ProPo per area (four in this case, to be used when the home L2 bank holds the ownership of the block and there are providers in the chip). No information about sharers is necessary in the home L2 bank thanks to the replacement mechanism described in Section 3.3.1.1, as it only transfers the ownership to the home L2 bank when there are no sharers left in the area. Only three of these four ProPos will ever be in use at the same time, but storing four is cheaper and easier than storing three, which also requires adding and managing area numbers. The directory information required in the L1 caches is a full-map bit-vector with a bit for each node in the area (to store the

Table 3.6: Leakage power of the caches per tile.

Protocol	Total Leakage Power (mW)	Difference with respect to directory	Tag Leakage Power (mW)	Difference with respect to directory
Directory	239		37	
DiCo	241	+1%	39	+5%
DiCo-Providers	222	-7%	20	-45%
DiCo-Arin	219	-8%	17	-54%

sharers when that L1 cache acts as the provider or the owner for the block) and one ProPo per area (to store the providers when the L1 cache acts as the owner). The L1C$ and L2C$ have the same size as in DiCo.

As for DiCo-Arin, when the home L2 bank holds the ownership for a block, the associated directory information consists of a full-map bit-vector of nta bits to store the sharers in the area along with $\log(na)$ bits to store the number of the area (where na is the number of areas, four in this case). However, if the block is shared between areas, the home L2 bank just needs one ProPo per area (to store providers to be communicated to requestors as local-area suppliers). Since the full-map bit-vector and the ProPos are never needed at the same time, only storage for the largest of them is actually needed (the former in this case). In the L1 cache, only a full-map bit-vector of nta bits is needed to store the sharers in the area. The L1C$ and L2C$ have the same size as in DiCo.

As a result, DiCo-Providers and DiCo-Arin reduce leakage power noticeably with respect to the flat directory, as can be seen in Table 3.6. The total leakage power of the caches is reduced by 8% in DiCo-Arin due to its smaller storage space for directory information, which is included in the tag structures of the tile. Overall, tags consume 54% less in DiCo-Arin than in the flat directory.

As the number of cores in CMPs grows, the effect of tag leakage power will increase, and the benefits of our proposals will be more noticeable. This can be seen in Table 3.7, which shows the storage overhead of the four cache coherence protocols for a range of number of processors and number of areas in the chip. For instance, with 1024 cores and 16 areas, DiCo-Arin reduces the overhead of the directory by 90.5%. The overhead of DiCo-Providers grows in general with the number of areas, as it needs one ProPo per area in the directory information. The overhead of DiCo-Arin goes down initially by increasing the number of areas, as the size of the full-maps for tracking VM-private data decreases. However, its overhead starts to grow again when the ProPos used to store the providers for shared-between-VM data start to dominate the directory storage space of the L2

Table 3.7: Storage overhead of the protocols depending on the number of cores and number of areas of the chip with respect to the tag and data arrays of the caches.

64 cores	**2 areas**	**4 areas**	**8 areas**	**16 areas**	**32 areas**	**64 areas**			
Directory	12.6%	12.6%	12.6%	12.6%	12.6%	12.6%			
DiCo	13.2%	13.2%	13.2%	13.2%	13.2%	13.2%			
DiCo-Providers	4%	5.1%	7.2%	10%	12.6%	12%			
DiCo-Arin	7.3%	4.5%	5.3%	6.6%	6.5%	2.3%			
128 cores	**2 areas**	**4 areas**	**8 areas**	**16 areas**	**32 areas**	**64 areas**	**128 areas**		
Directory	24.7%	24.7%	24.7%	24.7%	24.7%	24.7%	24.7%		
DiCo	25.3%	25.3%	25.3%	25.3%	25.3%	25.3%	25.3%		
DiCo-Providers	5%	6.2%	8.8%	13%	18.7%	24%	22.7%		
DiCo-Arin	13.4%	7.5%	6.8%	9.3%	12%	11.9%	2.5%		
256 cores	**2 areas**	**4 areas**	**8 areas**	**16 areas**	**32 areas**	**64 areas**	**128 areas**	**256 areas**	
Directory	48.9%	48.9%	48.9%	48.9%	48.9%	48.9%	48.9%	48.9%	
DiCo	49.6%	49.6%	49.6%	49.6%	49.6%	49.6%	49.6%	49.6%	
DiCo-Providers	6.7%	7.6%	10.6%	16.2%	24.8%	36.2%	47%	44.3%	
DiCo-Arin	25.5%	13.5%	8.5%	12.2%	17.4%	22.7%	22.7%	2.6%	
512 cores	**2 areas**	**4 areas**	**8 areas**	**16 areas**	**32 areas**	**64 areas**	**128 areas**	**256 areas**	**512 areas**
Directory	97.5%	97.5%	97.5%	97.5%	97.5%	97.5%	97.5%	97.5%	97.5%
DiCo	98.2%	98.2%	98.2%	98.2%	98.2%	98.2%	98.2%	98.2%	98.2%
DiCo-Providers	9.7%	9.7%	12.8%	19.6%	31.1%	48.5%	71.3%	92.9%	87.5%
DiCo-Arin	49.8%	25.7%	13.7%	15.2%	23%	33.6%	44.3%	44.3%	2.8%
1024 cores	**2 areas**	**4 areas**	**8 areas**	**16 areas**	**32 areas**	**64 areas**	**128 areas**	**256 areas**	**512 areas**
Directory	195%	195%	195%	195%	195%	195%	195%	195%	195%
DiCo	195.6%	195.6%	195.6%	195.6%	195.6%	195.6%	195.6%	195.6%	195.6%
DiCo-Providers	15.5%	13.1%	15.7%	23.3%	37.5%	60.8%	95.8%	141.7%	184.9%
DiCo-Arin	98.5%	50%	25.9%	18.6%	28.8%	44.6%	66.1%	87.6%	87.6%

cache. Remember that space is required only for the bigger of these two fields, as they never need to be stored at the same time.

Given a number of cores, an appropriate number of areas should be chosen for DiCo-Providers and DiCo-Arin to achieve a reasonably small overhead without losing other properties. A trade-off to consider is that using smaller areas implies that providers will be closer to the requestors, resulting in higher data locality and less network retransmissions upon prediction hits. On the other hand, smaller areas may make less likely the existence of providers in the local areas of requestors.

3.4.3 Dynamic Power Consumption

Figure 3.6 shows the total dynamic power consumption of all the protocols evaluated. Two kinds of workloads can be observed: those in which dynamic

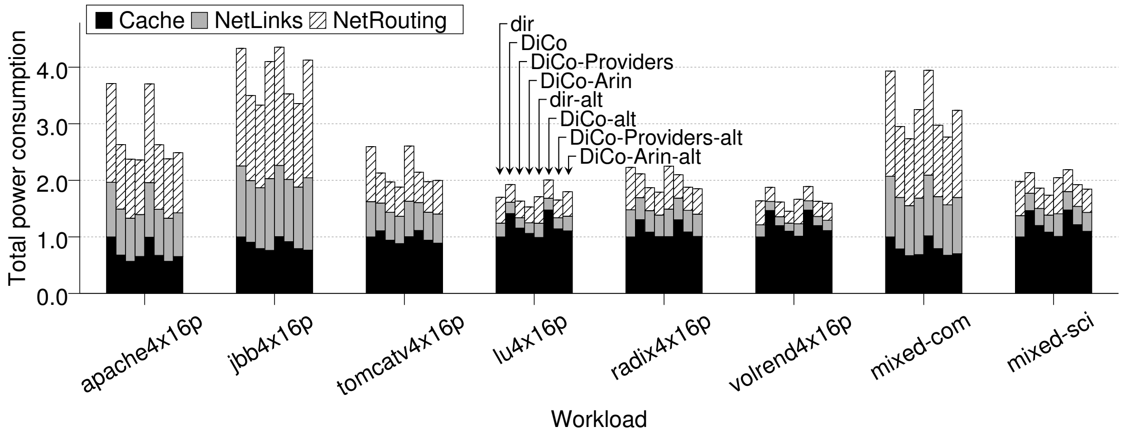


Figure 3.6: Total dynamic power consumption by protocol. Results normalized to the cache dynamic power consumption of the directory. Breakdown in cache, network links and network routing consumptions.

power consumption is dominated by L1 caches (tomcatv, lu, radix and volrend), and those in which it is dominated by L2 caches and network traffic (apache and jbb). L1-power-dominated workloads have working sets that fit in the L1 caches, and therefore very little traffic is generated and few directory accesses take place during their execution. On the other hand, L2-power-dominated workloads have working sets that are significantly larger than the L1 caches, which causes many L1 cache misses, resulting in higher network usage and number of directory accesses. L2-power-dominated workloads are the norm in real scenarios, as real applications have in general larger working sets that those of ordinary benchmarks [134]. We show both kinds of workloads for completeness in our analysis, as L1-power-dominated workloads are less suitable for the characteristics of DiCo-Arin and DiCo-Providers.

We can see in Figure 3.6 that our proposals reduce dynamic power consumption in every benchmark compared to the directory, but this reduction is especially noticeable in L2-power-dominated workloads. We find apache the most representative benchmark due to its large working set. Another benchmark with a large working set is jbb, but it also has a huge L2 cache miss rate (over 40%), making it a less realistic representative of the efficient operation of a consolidated server.

In general, our protocols reduce L2 cache and network power consumption, but at the same time they slightly increase L1 cache power consumption. DiCo

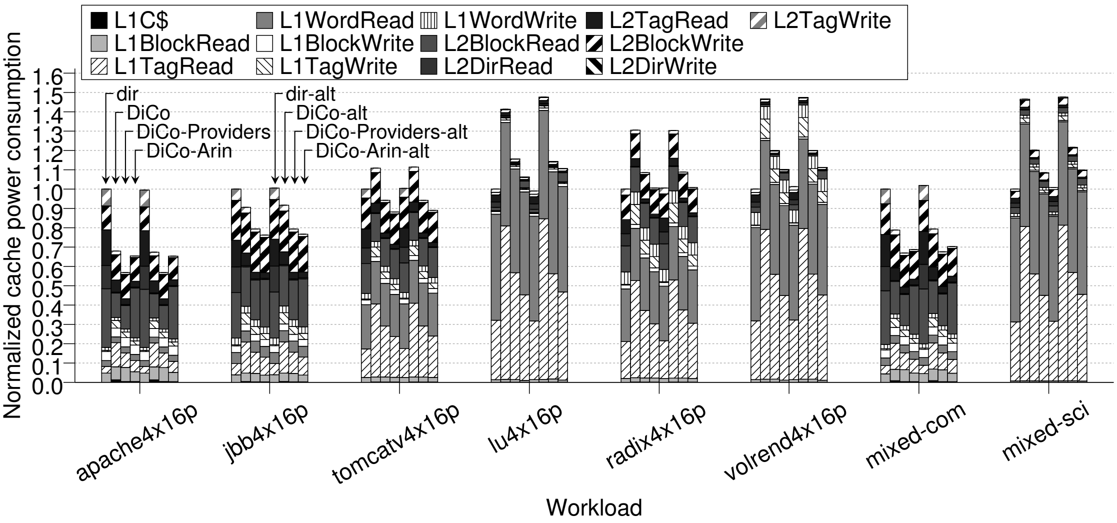


Figure 3.7: Normalized cache dynamic power consumption by protocol. Breakdown in cache events that cause the consumption.

noticeably reduces network usage with respect to the directory by solving many requests in just two hops. DiCo-Providers and DiCo-Arin reduce network usage even further thanks to the use of providers in the area for deduplicated data, shortening the average distance travelled by messages compared to DiCo.

Figure 3.7 breaks down the dynamic power consumption of the caches. This figure gives insight into what causes the different power footprint of the protocols, and in particular, puts these differences into context depending on the kind of benchmark. Due to the directory information stored in the L1 caches, tag accesses are more power consuming in DiCo-based protocols than in the flat directory. This causes DiCo-based protocols to use more power in the caches in some L1-power-dominated workloads (lu, radix and volrend with DiCo-Providers). Since network usage is low in these workloads, as evidenced in Figure 3.8 (and especially if compared to L1 cache usage), the savings in network traffic by our protocols can only improve the overall power consumption by a small margin on these workloads. Nevertheless, both DiCo-Providers and DiCo-Arin improve the original DiCo total power consumption by at least 10% in every L1-power-dominated workload.

The power consumption of the L1C$ of DiCo-based protocols is not a significant share of the overall power consumption. This is a result of the small size

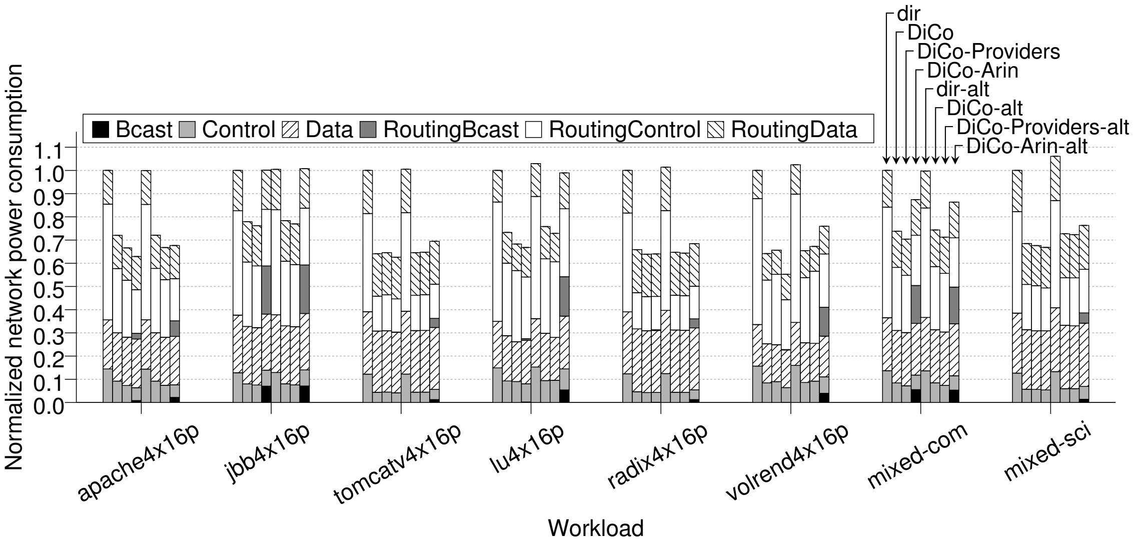


Figure 3.8: Normalized network dynamic power consumption by protocol. Breakdown in link usage and routing consumption.

of the L1C$ combined with the fact that it is seldom accessed, only on cache misses and when its contents are updated (by events such as invalidations or hint receptions). Other activities occurring as a consequence of the same events outweigh the power consumption of the L1C$ easily.

Regarding L2-power-dominated workloads, DiCo-Providers and DiCo-Arin reduce power consumption in apache by 38% with respect to the directory (Figure 3.6). This reduction comes from reductions in both network power and cache power (Figures 3.7 and 3.8). The power consumption generated by L2 cache tags is noticeably smaller in DiCo-Providers and even smaller in DiCo-Arin. In addition, L2 block reads, which are more energy consuming than L1 block reads, are more frequent in the directory since DiCo protocols often use an L1 cache as the provider to resolve misses in two hops.

Jbb represents the case in which pressure is highest in the L2 cache due to a huge working set. The L2 miss rate of jbb is over 40% for every protocol. We use it as the worst scenario for DiCo-Arin since this protocol uses more L2 space to store deduplicated data (increasing L2 pressure noticeably in this benchmark) and issues broadcasts to invalidate their L1 copies upon an L2 replacement. We can see in Figure 3.8 that broadcasts make DiCo-Arin network power consumption approach that of the directory in jbb. However, even in this worst-case workload, DiCo-Arin shows 4% less power consumption than the directory (Figure 3.6)

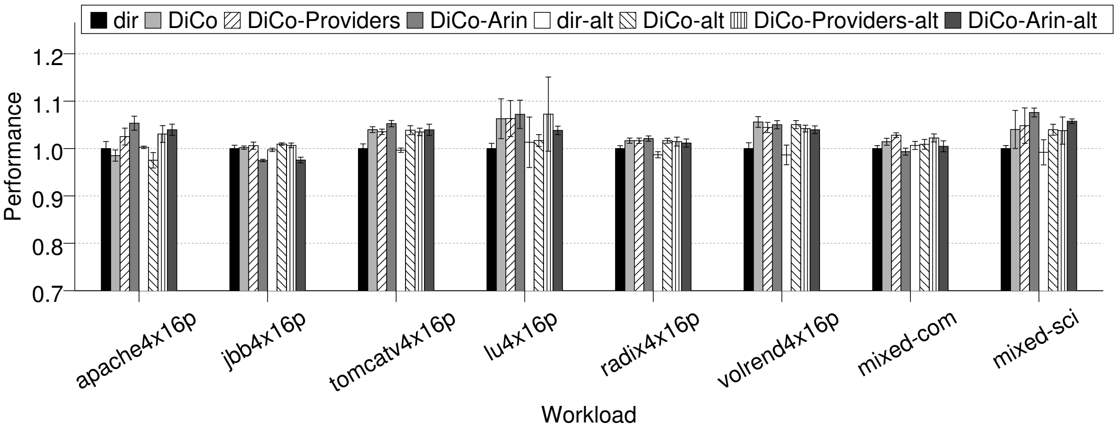


Figure 3.9: Performance (bigger is better).

thanks to the lower use of distant L2 caches due to the operation of DiCo that resolves many misses in two hops in L1 caches. DiCo-Providers proves the most reliable protocol in terms of power consumption and also reduces total power consumption in jbb by 22% with respect to the directory.

Regarding the alternative configuration in which the VMs do not match the areas, no significant differences are observed with respect the optimal configuration in which the VMs fit the areas, beyond the logical increment in broadcast traffic in DiCo-Arin due to the extra invalidations of read/write blocks that in this configuration are shared between areas. Nevertheless, despite this traffic increment, the power consumption of DiCo-Arin keeps being smaller than that of the directory. In addition, experiments with parallel applications that make use of all cores also show results in line with these, proving the general applicability of DiCo-Providers and DiCo-Arin.

3.4.4 Performance Results

The performance results of the protocols can be found in Figure 3.9. The main conclusion is that DiCo-Providers and DiCo-Arin show no significant degradation compared to the original DiCo. In the most representative benchmark (apache), DiCo-Providers and DiCo-Arin outperform the directory by 3% and 6% respectively. This is an expected result as DiCo-Providers and DiCo-Arin increase data locality. Only in jbb does DiCo-Arin perform 2% worse than the highly optimized

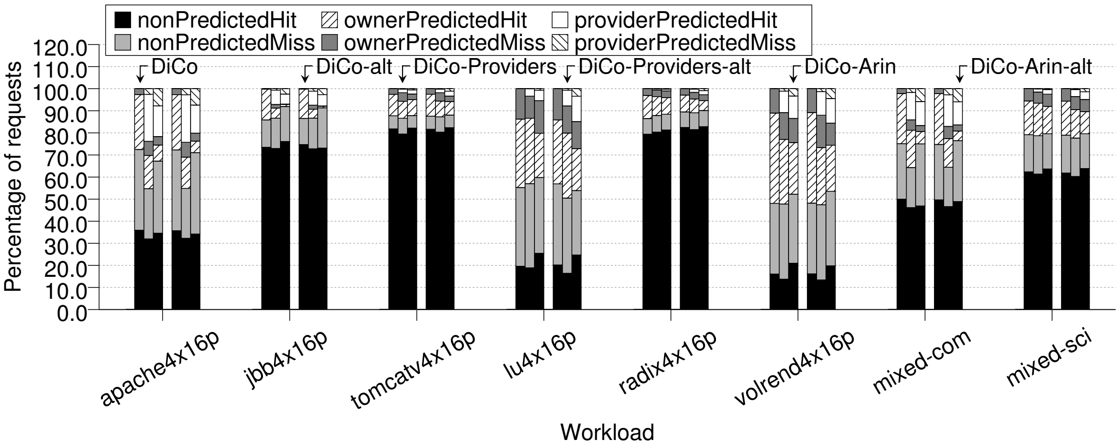


Figure 3.10: Breakdown of L1 misses in six categories depending on whether they were predicted or not, on whether the destination was an owner or a provider in the area, and on whether the prediction succeeded or not.

flat directory, which is caused by the particular L2 miss rate characteristics of jbb pointed out earlier that make it the worst scenario for DiCo-Arin.

In Figure 3.10 we can see that, in some benchmarks, a significant percentage of the requests can be resolved by predicting the provider. In the case of apache, 21% of all the requests are resolved in this way in DiCo-Providers. Taking into account that the theoretical average distance between any two nodes in a 2D mesh is $\frac{2}{3}\sqrt{ntc}$ (where ntc is the number of tiles in the chip), a two hop miss in our 64-tile CMP (with arbitrary origin and destination) would traverse 10.6 links on average if solved in two hops, and 16 links on average when incurring directory indirection. The misses that hit in the provider only take two hops inside a 16-tile area. This results in a theoretical average of 5.4 links traversed to resolve such misses, instead of the 10.6 links needed in DiCo and 16 links with directory indirection. This matches our experimental results, which reveal a reduction in number of links traversed with respect to DiCo of 38% and 40% for DiCo-Providers and DiCo-Arin in this kind of misses, respectively. We call these misses resolved inside the area *shortened misses*. Overall, shortened misses cause a noticeable reduction in the average miss latency and power consumption. As the number of tiles and VMs increases, this benefit grows. For example, in a densely virtualized 256-tile CMP with 4-tile areas (that is, 64 VMs), indirect misses would take an average of 32 links, 2-hop misses would take 21.3 links,

Table 3.8: Links traversed per miss for misses solved on a remote L1 cache depending on the number of cores and number of areas of the chip for each protocol.

64 cores	**2 areas**	**4 areas**	**8 areas**	**16 areas**	**32 areas**	**64 areas**			
Directory	16	16	16	16	16	16			
DiCo	10.7	10.7	10.7	10.7	10.7	10.7			
DiCo-Providers	8	5.3	4	2.6	2	10.7			
DiCo-Arin	8	5.3	4	2.7	2	10.7			
128 cores	**2 areas**	**4 areas**	**8 areas**	**16 areas**	**32 areas**	**64 areas**	**128 areas**		
Directory	24	24	24	24	24	24	24		
DiCo	16	16	16	16	16	16	16		
DiCo-Providers	10.7	8	5.3	4	2.7	2	16		
DiCo-Arin	10.7	8	5.3	4	2.7	2	16		
256 cores	**2 areas**	**4 areas**	**8 areas**	**16 areas**	**32 areas**	**64 areas**	**128 areas**	**256 areas**	
Directory	32	32	32	32	32	32	32	32	
DiCo	21.3	21.3	21.3	21.3	21.3	21.3	21.3	21.3	
DiCo-Providers	16	10.7	8	5.3	4	2.7	2	21.3	
DiCo-Arin	16	10.7	8	5.3	4	2.7	2	21.3	
512 cores	**2 areas**	**4 areas**	**8 areas**	**16 areas**	**32 areas**	**64 areas**	**128 areas**	**256 areas**	**512 areas**
Directory	48	48	48	48	48	48	48	48	48
DiCo	32	32	32	32	32	32	32	32	32
DiCo-Providers	21.3	16	10.7	8	5.3	4	2.7	2	32
DiCo-Arin	21.3	16	10.7	8	5.3	4	2.7	2	32
1024 cores	**2 areas**	**4 areas**	**8 areas**	**16 areas**	**32 areas**	**64 areas**	**128 areas**	**256 areas**	**512 areas**
Directory	64	64	64	64	64	64	64	64	64
DiCo	42.7	42.7	42.7	42.7	42.7	42.7	42.7	42.7	42.7
DiCo-Providers	32	21.3	16	10.7	8	5.3	4	2.7	2
DiCo-Arin	32	21.3	16	10.7	8	5.3	4	2.7	2

and shortened misses would take just 2.6 links. Table 3.8 shows typical values of traversed links to solve cache misses for a number of combinations of core count and number of areas. These values, combined with the overhead values shown in Table 3.7, can be used as a guideline for selecting configurations with good trade-offs for a given CMP size.

Figure 3.9 also shows that using the alternative configuration of VMs does not produce any significant changes in the performance of the VMs in any of the protocols. Such absence of variation could be expected for the directory and DiCo, for which different placements of the VMs do not make much of a difference. On the other hand, for DiCo-Providers and DiCo-Arin, the absence of a larger penalty in performance may not be so intuitive. Two reasons explain why DiCo-Providers and DiCo-Arin keep performing well. First, when a VM executes in cores of more than one area, the L1 owners are still located within the VM and are often accessed in two hops, increasing locality just like with ideal VM placement. Second, since the VM executes in more than one area, data private to the VM (in addition to data shared between the VMs) can benefit from multiple providers to increase data locality further than when using just the owner. These private data may be supplied by a provider in the area of the requestor, which is closer to the requestor than the owner when it is located in another area. This way, the overall performance and prediction accuracy of our proposals remain stable even if the VMs execute in several areas or if parallel applications use all areas.

3.5 Related Work

Scalable and power-aware cache coherence has gained interest in recent years. The Tagless Coherence Directory [180] reduces the coherence storage overhead compared to a flat directory by using bloom filters to store coherence information, reducing power consumption. Our proposals are orthogonal to the Tagless Coherence Directory, and they could be combined with the use of bloom filters to further reduce the size of coherence information.

TurboTag [117] uses bloom filters to avoid unnecessary tag lookups and reduce power consumption. Again, our proposals are orthogonal to TurboTag.

Coherence protocols have also been adapted to take advantage of heterogeneous networks to reduce power consumption by transmitting critical and short messages through fast power-consuming wires and non-critical messages through slower low-power wires [54]. Instead, our proposals increase data locality to reduce the number of retransmissions in the electronic network. Nevertheless, both strategies can be jointly applied to server consolidation.

SARC Coherence [98] reduces network traffic by using tear-off copies of the block in a weak-ordered memory system and by using prediction to avoid the indirection of the directory. Our proposals use DiCo as a baseline to avoid directory indirection just like SARC Coherence. In addition, our proposals take prediction further, to show that it is especially beneficial when close providers, predicted upon cache misses, are allowed to supply a copy of the block.

As for virtualization, a number of proposals address scalability or performance, but up to our knowledge ours is the first one to address power consumption. Coherency Domains (CDs) have been proposed to increase the scalability of cache coherence in scenarios such as server consolidation [23, 101, 140]. They isolate the coherence transactions of different CDs, each comprising a set of the processors of the chip, preventing different VMs that execute in different CDs from interacting with each other. Moreover, the data in the LLC cache is stored closer to the cores that use them, since each coherency domain is given a private LLC cache. However, contrary to our proposals, CDs do not allow the simultaneous use of all the resources of the chip for a single task. Tasks are confined to the static coherency domains defined by the design on the chip, requiring modifications to the operating systems and hypervisors to operate in such environment. In addition, CDs do not enable memory deduplication since each CD is given its own independent share of the physical memory and no coherence is kept for data between different domains. This also involves that the benefits of deduplication cannot be translated to the cache level either. Coherence Domains miss these opportunities that are paramount on densely consolidated servers.

Another coherence proposal for server consolidation is Virtual Hierarchies (VHs) [128]. VHs also achieve isolation among VMs, and the dynamic nature of the cache hierarchy additionally allows for the dynamic allocation of cache resources to VMs. However, contrary to our proposals, VHs increase the overhead and power consumption of cache coherence due to a second level of coherence information introduced to enable the dynamic behaviour of the hierarchy. Both levels require full-map bit-vectors (one bit per core). Alternatively, a second level using single-bit entries in DRAM reduces the extra overhead of VHs over a flat directory, but this scheme requires issuing broadcast messages to locate (not just invalidate) any block tracked by this second level directory. In any case, VHs do not focus on optimizing energy efficiency. Furthermore, VHs reduplicate previously deduplicated data in the shared levels of the cache hierarchy, which also results in an increase of the LLC miss rate [61], missing the opportunity to bring the benefits of memory deduplication to the cache level.

Mechanisms at the level of the interconnection network have also been proposed to isolate the traffic of each VM [53]. These mechanisms do not tackle the overhead of cache coherence. Our proposals achieve isolation by placing virtual machines within different areas and resolving cache misses by means of the local owner and providers. In addition, they reduce the overhead of cache coherence and enable gains derived from memory deduplication at the cache level.

3.6 Conclusions

Server consolidation is gaining importance as the number of cores integrated in a single chip grows. The number of virtual machines per server is also likely to increase to take advantage of such a number of cores. We have proposed a new scheme with the chip statically divided in areas in which deduplicated data is stored only once in the shared level of cache and yet data locality is improved thanks to the use of *providers* close to the requestors. We have proposed two different protocols based on this scheme: DiCo-Providers and DiCo-Arin. DiCo-Arin is simpler than DiCo-Providers and it requires less hardware for storing sharing information. On the other hand, DiCo-Arin relies on broadcast to invalidate data shared between areas (i.e., deduplicated data), which makes it less flexible for general use.

We have shown that our protocols achieve a 59–64% reduction in directory information in cache for a 64-tile CMP with just 4 VMs, which reduces static power consumption by 45–54% and improves scalability. They reduce dynamic power consumption by up to 38% for the most representative workload (apache). When the weak points of our protocols are tested with non-realistic scenarios in which little network traffic is generated and few L1 cache misses take place, the power consumption of our protocols is still lower than that of the optimized directory.

Our proposals are flexible enough to still provide most of their advantages even with suboptimal placement of VMs —not matching the areas—, as well as with parallel applications that use all the cores.

Our protocols do not show any significant degradation in performance with respect to a directory protocol, even if the placement of the VMs does not exactly match the static areas. On the contrary, they noticeably outperform the directory in most workloads thanks to the use of providers, with speedups up to 6% with respect to the optimized directory protocol in apache. We also expect that as virtualization density increases, with tens of virtual machines running in a single server, the advantages of our proposals will become even more prominent.

Chapter 4

Distance-Aware Partially Shared Cache Organizations

As the number of cores in tiled CMP proposals increases, they often assume a partially shared last level cache (LLC), since it provides a good trade-off between access latency and cache utilization. In this chapter, we describe a novel way to statically map memory addresses to LLC banks that takes into account the average distance between the banks and the tiles that access them in order to increase data proximity. Contrary to traditional approaches, our mapping does not group tiles in clusters within which all cores access the same cache banks. Instead, two neighboring cores access different sets of closer banks, minimizing the average distance travelled by cache requests and responses. Results for a 64-core CMP show that our proposal improves both execution time and the energy consumed by the network by 13% when compared to traditional mappings. Moreover, our proposal comes at a negligible cost in terms of extra hardware and its benefits in terms of both energy consumption and execution time increase with the number of cores.

4.1 Background

As discussed in the introduction of this thesis, tiled-CMPs contain a last-level cache (LLC) that, although physically distributed among tiles, can be logically organized in a number of ways. The organization of the LLC is paramount to avoid costly off-chip accesses [115], as well as to optimize on-chip communication

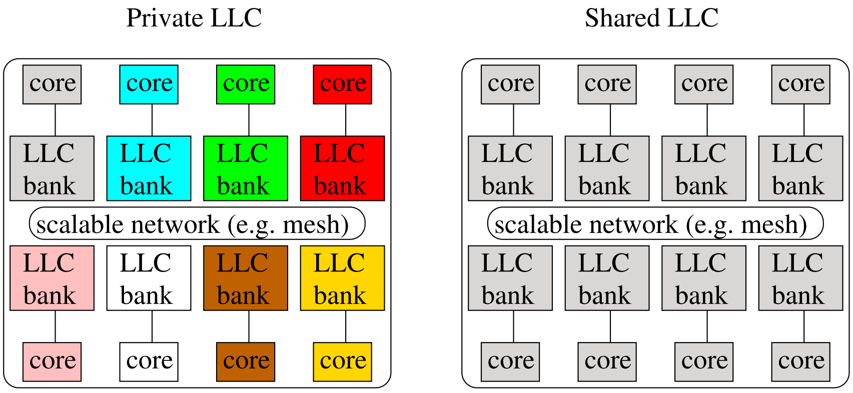


Figure 4.1: Examples of private (left) and shared (right) last level caches.

and cache utilization. Possible organizations range from a completely shared LLC [80,95,100,105], in which all the cache banks of the chip can be accessed by any core, to private LLCs [1,7], in which each core accesses its own private cache bank. Figure 4.1 depicts these two extreme designs. In addition, intermediate designs are possible, resulting in partially shared cache organizations [75,87,135], with several cores sharing their cache resources. We will use the term *sharing degree* to denote the number of cores that can access each cache bank (i.e., it is 1 for private LLC organizations and it is equal to the total number of cores in a fully shared LLC organization). Equivalently, the sharing degree is also the number of different banks that can be accessed by each core. Figure 4.2 shows two partially shared LLCs with different sharing degrees. The optimal sharing degree depends on parameters such as network latency, main memory bandwidth and latency, cache size and applications' working set. Most proposals for CMPs with large core counts assume at least some degree of sharing in the LLC.

All these LLC configurations usually work with a static mapping. This means that the home LLC bank for each block is decided at design time and never changes. Usually, a subset of the bits of the block address (known as bank selector) determines the home LLC bank for the block. In a shared cache, all cores use the same mapping based on the bank selector. In a partially-shared cache, each cluster uses a different mapping (directing the requests to the LLC banks of the cluster). In a private cache, the mapping is always to its local LLC bank (no bank selector is needed in this case).

In a CMP with a shared or partially shared LLC organization, the time and energy that a request and its response spend in the interconnection network are a significant part of the total time and energy spent accessing the LLC. Both time

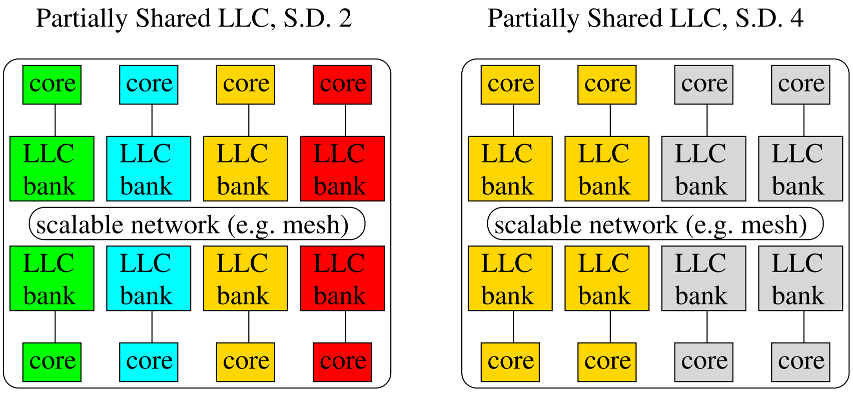

Partially Shared LLC, S.D. 2
Partially Shared LLC, S.D. 4
core
LLC bank
scalable network (e.g. mesh)

Figure 4.2: Examples of partially-shared last level caches. Sharing degree of 2 (left) and 4 (right).

and energy depend on the distance between the core that carries out the access and the particular LLC bank that holds the data.

Logically, increasing the sharing degree increases the average distance that messages travel, because some of the LLC banks will have to be further from the core. Hence, as the number of cores in CMPs increases, fully shared organizations become less attractive in terms of access latency and network usage. However, some degree of sharing is still desirable, as it improves LLC utilization because shared caches allocate a single copy of each shared data for all cores, contrary to private caches, which replicate shared data in the private caches of every core that accesses the data. This replication increases cache pressure and results in extra LLC misses, which in turn increase off-chip traffic. Also, lower sharing degrees limit the number of banks that can be used by individual cores, generating many off-chip accesses when some cores have working sets that do not fit in their accessible LLC banks. As a worst case example, consider a single-thread application running alone on a chip with private LLCs. In this case, there could be many off-chip accesses even if most of the cache resources of the chip were unused. Since off-chip bandwidth increases more slowly than the number of cores in a CMP [154], avoiding LLC misses becomes even more important as the number of cores increases. Hence, partially shared organizations provide a good trade-off between the fast latency of private caches and the improved capacity utilization of shared caches.

The most usual way to organize partially shared organizations is by dividing the CMP in clusters of tiles which share their LLC banks among them (like in Figure 4.3). The mapping of memory blocks to LLC banks in the cluster is determined by $log_2(s)$ bits of memory addresses, where s is the sharing degree. By using this static mapping, the sets of LLC banks accessed by cores in two different clusters do not overlap. Additionally, a directory is commonly used to enforce cache coherence among clusters.

The above organization does not take into account the distance between a core and the cache banks that the core accesses. In fact, there are very significant differences between cores within the same cluster, because the cores in the center of clusters have the LLC banks that they access nearer (on average) than the cores near the edges of the clusters (as shown in Figure 4.6). Other LLC organizations have been proposed (see Section 4.4), including dynamic mappings.

The proposal introduced in this chapter, DAPSCO (Distance-Aware Partially Shared Cache Organization), aims at improving partially shared cache access. We present it in the context of last-level caches, but it can be applied to any partially shared level of the cache hierarchy. Some basic notions about DAPSCO, introduced at this point, will help put it in perspective with the background provided in this section. DAPSCO uses a static block mapping policy that optimizes the average distance to access remote LLC banks to improve data proximity, reducing the energy consumption and increasing the overall performance of the system. This is possible because, differently from other static mappings, DAPSCO mappings do not group tiles in clusters. Instead, each LLC bank serves a portion of the memory space to its neighboring cores, and every core accesses a different set of LLC banks.

Since DAPSCO employs a static mapping, it does not need any extra power-consuming structures or mechanisms with respect to traditional mapping policies for partially shared caches. Hence, the design of DAPSCO is not a trade-off between energy and performance. Instead, both execution time and energy consumption are improved, increasing the energy efficiency of the system notably.

Finding the optimal block mapping that minimizes the number of links traversed for a given on-chip network topology is an intractable problem for large CMPs, and therefore, we have used a greedy algorithm to find near optimal block mappings for both mesh and torus topologies. It is worthy to note that this algorithm is only employed at design time to obtain the static mapping function for each CMP configuration. At runtime, only the static mapping found by the algorithm, hard-wired in the chip, is used instead of the traditional mapping.

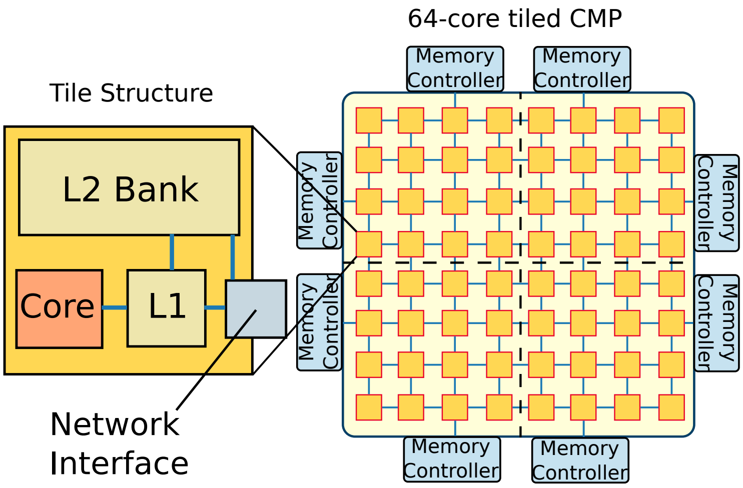


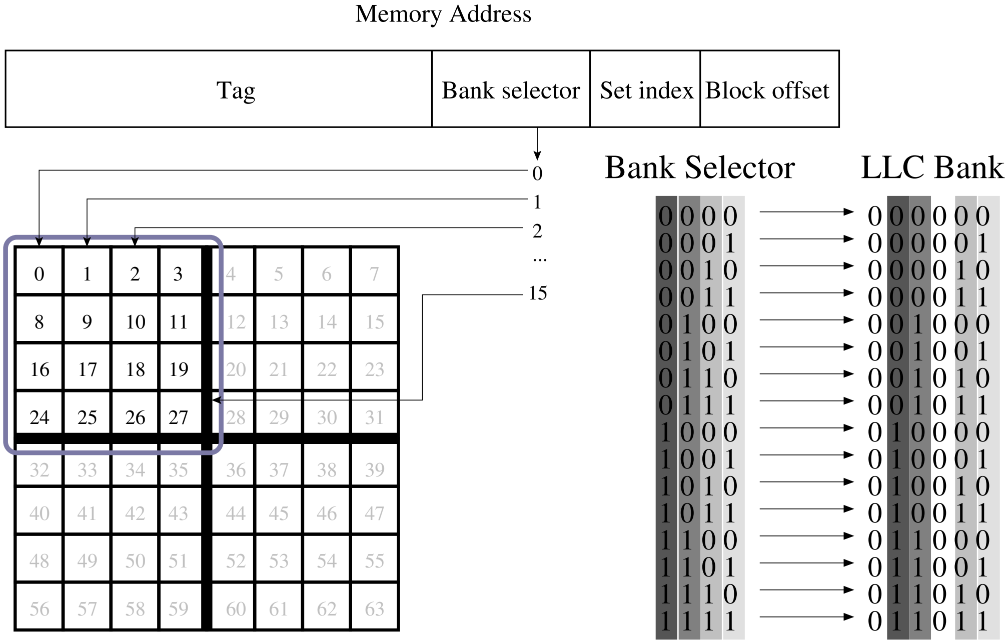
Figure 4.3: Diagram of a 64-core tiled CMP with a sharing degree of 16. The dashed lines separate the 4 clusters of tiles which share LLC banks among them.

4.1.1 Base Architecture

The base architecture targeted by our proposal is the tiled-CMP described in Section 1.4, in which the chip is assembled by replicating basic building blocks named tiles. In this chapter, we assume a partially shared last-level cache formed by clusters of cores. The cores inside a cluster share their LLC banks —L2 banks—, as indicated with dashed lines in Figure 4.3. The L1 and L2 caches are non-inclusive. An optimized MOESI, directory-based cache coherence protocol is used. We assume 64 tiles through the chapter for all examples and for the evaluation section, although our proposal would achieve higher benefits with a larger number of cores. In the rest of the chapter, we call this design “traditional partially shared cache organization”.

As mentioned before, the *sharing degree* of the cache determines the number of cores that access each cache bank (which is equal to the number of banks accessed by each core) and, in the case of traditional partially shared caches, this also determines the size of the clusters of tiles that share the same cache banks. We always assume the same sharing degree for every LLC bank (i.e., all LLC banks are accessed by the same number of cores) for the sake of fairness in the pressure exerted over them.

In this partially shared design, the cluster number is combined with certain bits of the memory address of each block (bank selector) to determine which tile of the chip contains the home cache bank for the block. Figure 4.4 shows how



Bank Selector	LLC Bank
0000	000000
0001	000001
0010	000010
0011	000011
0100	001000
0101	001001
0110	001010
0111	001011
1000	010000
1001	010001
1010	010010
1011	010011
1100	011000
1101	011001
1110	011010
1111	011011

Figure 4.4: Bank selection in a partially shared LLC for the base architecture with a sharing degree of 16. The 16 cores within each of the four clusters share their LLC banks among them. The numbers on the tiles correspond to the tile identifiers. We show the bit correspondence used by the cores in the marked cluster, cluster number 00. By combining the bank selector and the cluster number (00), the identifier is generated of the tile whose LLC bank stores the block.

the cores of the upper left cluster combine the bank selector bits of an address with the cluster number to determine the LLC bank containing the block. For instance, in cluster 0 (00), bank selector 15 (1111) is mapped to bank 27 (011011). The correspondence between all the possible values of the bank selector and the right home cache banks of the cluster is also shown in the figure. Notice the shades under the bit values that indicate the equivalence between bits in the bank selector and bits in the cache bank identifier. Non-shaded values are determined by the cluster number (00 in the example). This correspondence works in such a way that the cores within the cluster access the banks within the cluster.

4.1.2 Coherence Issues

In order to keep cache coherence, every time that some data is transferred from a shared cache level to a lower private level, resulting in the generation of copies of the block, the shared level needs to track these private copies. The total overhead of directory information required to do this depends on a combination of the size of the caches whose contents must be tracked and their sharing degree (i.e., the number of possible copies). Usually, L2 caches are several times larger than L1 caches; hence the directory needed for tracking private copies when the L2 caches are private is several times bigger than the one needed for tracking the copies stored in private L1 caches when the L2 cache is shared.

In the case of partially shared LLCs, two levels of coherence are needed. First, each LLC bank is shared by a number of private caches, and the copies of the blocks stored in these private caches must be tracked to provide coherence inside the cluster. Second, since the LLC is partially shared, the blocks stored in the LLC banks are shared within the same cluster but private with respect to another cluster. Therefore, these blocks must also be tracked to provide coherence between clusters.

We assume a directory in the L2 caches for the first level of coherence —intra cluster— and a directory cache at the memory controller for the second level of coherence —inter cluster—. Eight memory controllers are placed along the edges of the chip and memory addresses are interleaved across them.

However, having several levels of coherence information does not involve an increase in the total directory overhead compared to a completely private or completely shared LLC, which only requires one level. On the contrary, the directory overhead can be greatly reduced by using two or more levels, and for this reason this hierarchical coherence architecture in particular has been proposed as a good design to overcome the scalability issues of cache coherence [125]. For instance, let us assume a 64-core tiled CMP with L1 caches, L2 caches and main memory. If the L2 cache is shared among all the cores, 64-bit vectors are needed to track the private copies of the blocks stored in the L1 caches. On the other hand, if the L2 banks are private to the cores, 64-bit vectors are needed again, this time to track the private copies stored in the L2 banks. However, if a partially shared L2 with a sharing degree of 8 is used, 8-bit vectors are needed to track all the possible copies in the eight L1 caches that make up a cluster, and 8-bit vectors are also needed to track all the possible copies in the L2 banks of eight different clusters, one bank per cluster, that can hold private copies of the same block. Smaller entries result in a smaller total overhead.

Table 4.1 shows the overhead introduced to store the sharing vectors in chips with different numbers of cores and sharing degrees, assuming in all cases a 4× over-provisioned sparse directory [69] and L2 banks that are eight times bigger than the L1 banks. The smallest overhead is always found in an intermediate

Table 4.1: Sharing information overhead of partially shared caches on the total cache capacity of the chip.

	Sharing Degree									
Cores	1	2	4	8	16	32	64	128	256	512
64	44%	22%	11%	6.3%	4.2%	4.2%	5.6%			
128	89%	45%	23%	12%	7%	5.6%	6.9%	11%		
256	178%	89%	45%	23%	13%	8.3%	8.3%	13%	22%	
512	356%	178%	89%	45%	24%	14%	11%	14%	24%	44%

partially shared organization, with the overhead growing notably as we move toward a private LLC (sharing degree of one) or toward a shared LLC (sharing degree equal to core count). In general, the overhead of the multi-level directory for partially shared caches grows with $\sqrt{n}$, where n is the number of cores, rather than linearly with n as the overhead of the flat directory typically used for shared or private LLCs. This is another reason that makes partially shared caches a very interesting alternative to increase cache coherence scalability. Although scalable hierarchical directories can be potentially used in combination with any cache hierarchy, partially shared caches are most convenient because they simplify the coherence protocol implementation greatly, as their block storage is organized in parallel to the directory hierarchy.

4.1.3 Block Mapping and Distance to Access the LLC

The LLC acts as a last barrier to prevent costly requests to main memory. In order to attain good performance, the miss ratio of the LLC must be low. In addition, to achieve good performance in a two level cache hierarchy with a partially shared LLC, most L1 cache misses are expected to be resolved by retrieving the requested data within the cluster. To illustrate this, Figure 4.5 shows the breakdown of L1 cache misses in three categories for a number of workloads executed on the machine described in the evaluation section (see Table 4.3 for details). We consider that a hop is the sending of a message from a particular origin to a particular destination. Depending on the number of hops involved, we differentiate three kinds of L1 cache misses:

2-hop misses. A request message for the accessed block is sent from the L1 cache to the home L2 cache bank determined by the block mapping policy (first

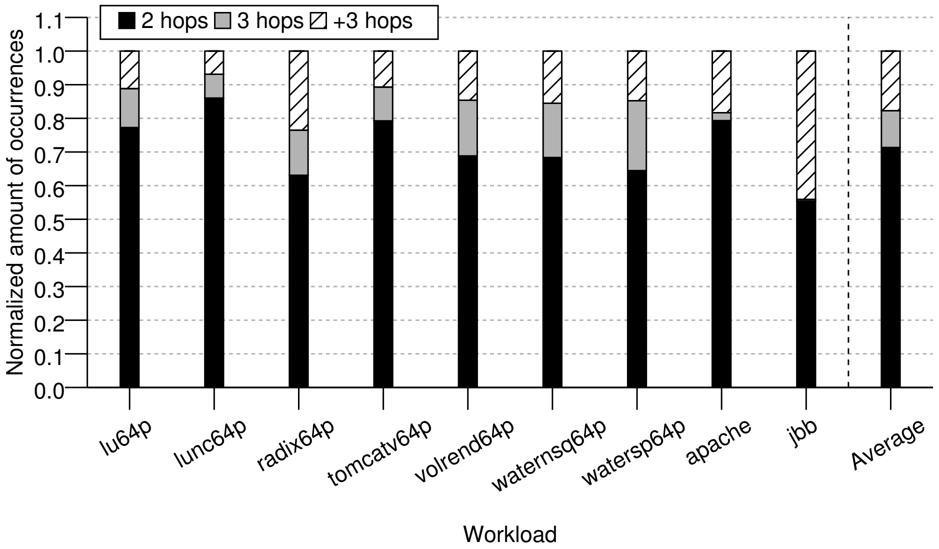


Figure 4.5: Cache miss breakdown.

hop). The block is found in that L2 bank, which sends it in a response message (second hop).

3-hop misses. A request message for the accessed block is sent from the L1 cache to the home L2 cache bank determined by the block mapping policy (first hop). There is no valid copy of the block in the L2 bank, but directory information is found indicating that there is a valid copy in an L1 cache. The request message is forwarded to that L1 cache (second hop). Upon the reception of the forwarded request, the L1 cache sends a message containing a copy of the block to the requestor (third hop).

+3-hop misses. A request message for the accessed block is sent from the L1 cache to the home L2 cache bank determined by the block mapping policy. There is no valid copy of the block in the L2 bank, and the directory information indicates that there are no valid copies in the set of L1 caches of the cluster. The request message is forwarded to the memory controller. This kind of miss requires at least four hops (in the case that the block is retrieved from memory) or even more hops if another L2 bank must be accessed to find a valid copy. For simplicity, we do not further divide +3-hop misses in sub-types in our analysis. In general, just a small percentage of these require a memory access.

Of these types, the first two are the most beneficial for performance. The main conclusion drawn from Figure 4.5 is that most L1 misses belong to the 2-hop miss category, matching the efficient behavior expected from a partially-shared cache. In 2-hop misses, the traversal of links to and from the LLC accounts for most of the latency to resolve the misses. In no single benchmark do 2-hop misses represent less than 50% of the total, and on average, they account for over 70% of misses. Jbb is the benchmark with the smallest amount of 2-hop misses (55%), and it is due to the huge L2 miss rate incurred by this workload, over 40%, caused by its enormous working set. 3-hop misses also represent a significant fraction of the total number of L1 cache misses of some workloads, and these misses are significantly affected by the distance to the home LLC bank too.

Therefore, the average latency of L1 cache misses is mainly determined by the distance from the requesting core to the L2 bank in which the data is located. As a consequence, the mapping of blocks to cache banks is key for improving the performance of the cache hierarchy of a CMP, since it determines the distance from a core to the LLC banks that it accesses.

The clustered design of traditional partially shared caches is not particularly suited to provide short distances from the cores to the LLC banks that they access. Making each cluster of cores share their LLC banks results in many cache accesses with high latency, especially for those cores that are located far from the center of the cluster. To illustrate this, Figure 4.6 shows the average distance, measured in number of links traversed, from each core to the LLC banks it accesses in the traditional partially shared cache organization that we take as a baseline in this chapter, for 64 cores and a sharing degree of 16.

We can see in Figure 4.6 that, as we move towards the edges of the clusters, the average distance from a core to the LLC banks of the cluster grows, increasing the energy and latency required by LLC accesses. As the number of cores and sharing degree increase, the differences among tiles become larger.

This kind of partially shared cache organization made sense when clusters of cores were located in different chips, because accessing the LLC banks within the chip is faster than accessing banks in another chip. However, with large CMPs in which a large number of cores are located in the same die, a partially shared cache organization like this is suboptimal.

Example calculation

2	1	2	3
1	0	1	2
2	1	2	3
3	2	3	4

Aggregate: 32 links
Average: 2 links

3	2.5	2.5	3	3	2.5	2.5	3
2.5	2	2	2.5	2.5	2	2	2.5
2.5	2	2	2.5	2.5	2	2	2.5
3	2.5	2.5	3	3	2.5	2.5	3
3	2.5	2.5	3	3	2.5	2.5	3
2.5	2	2	2.5	2.5	2	2	2.5
2.5	2	2	2.5	2.5	2	2	2.5
3	2.5	2.5	3	3	2.5	2.5	3

Figure 4.6: Average distance in number of links from each core to the LLC banks that it accesses in a 64-tile CMP with a sharing degree of 16 (four clusters).

4.2 Distance-Aware Partially Shared Cache Organizations

Any (partially) shared LLC organization based on a static mapping is defined by two elements. The first one is the access relationships between cores and LLC banks: each core accesses s LLC banks and each LLC bank is accessed by s cores, where s is the sharing degree. The second one is the labeling of LLC banks. Each LLC bank stores data corresponding to one of the s portions of the memory space, which is determined by its *bank label*. In addition, these two elements combined must ensure that every core accesses one LLC bank for each bank label, so that the core can access the whole memory space through the LLC.

Our observation is that, by allowing each core to access a different subset of the LLC banks (the most appropriate given the core location), we can optimize the distance from each core to its LLC banks. To achieve this, the mapping function that is used for finding the home LLC bank of each block will be different for each core (in contrast to a traditional partially shared organization, where all the cores of the same cluster share the same mapping).

Unfortunately, finding a good LLC organization is not an easy task, as we have to make sure that it operates properly. For instance, a trivial mapping in which each core accesses its closest banks would not allow cores to access one LLC bank for each bank label, resulting in an inoperative chip. To overcome this difficulty, we formalized the problem and developed a way to explore the search space of valid configurations.

Formally stated, a partially shared LLC organization is represented by a labeled balanced directed graph where each vertex represents a tile in the CMP and each arc represents an access relationship between a core (tail of the arc) and an LLC bank (head of the arc). Each arc has a weight that corresponds to the number of links traversed to go from the source tile to the destination tile. The weight of an arc depends on the placements of the core and the LLC bank connected by the arc, as well as on the particular underlying network topology that connects the tiles. The directed graph is also balanced, because all cores access the same number of LLC banks and all LLC banks are accessed by the same number of cores. Therefore, the indegrees and outdegrees of every vertex are equal to the sharing degree, s, which must be a divisor of the number of vertices, $|V|$. Each vertex has an associated label, existing $|V|/s$ vertices labeled with each of the numbers between 0 and $s-1$. These numbers are the bank labels of the LLC banks, representing which portion of the memory space is mapped to the LLC bank of the tile. In addition, two arcs with the same tail (core) cannot have heads (accessed LLC banks) with the same bank label, because each core needs to access one and only one LLC bank for each possible bank label in order for the LLC organization to work properly.

To find the best possible partially shared LLC organization, we must search for a graph that has the properties stated above and, at the same time, minimizes the total sum of the weights of its arcs (i.e., the distance from the cores to the LLC banks). The resulting cache organization will improve the performance of the system and will reduce the energy consumption of the network. We call these configurations of the chip “Distance-Aware Partially Shared Cache Organizations” (DAPSCO). DAPSCO does not increase the complexity of the coherence protocol nor the coherence information overhead compared to traditional partially shared cache organizations, as we show in Section 4.2.3.

Finding the optimal DAPSCO for a given CMP size and sharing degree is an NP-complete problem. In Section 4.2.1, we explain how to use heuristic algorithms to find near optimal configurations. However, finding the optimal DAPSCO is easier for CMPs with symmetric interconnects such as torus-based networks, and a method for doing so in which every core uses the same pattern to access the LLC will be explained in Section 4.2.2.

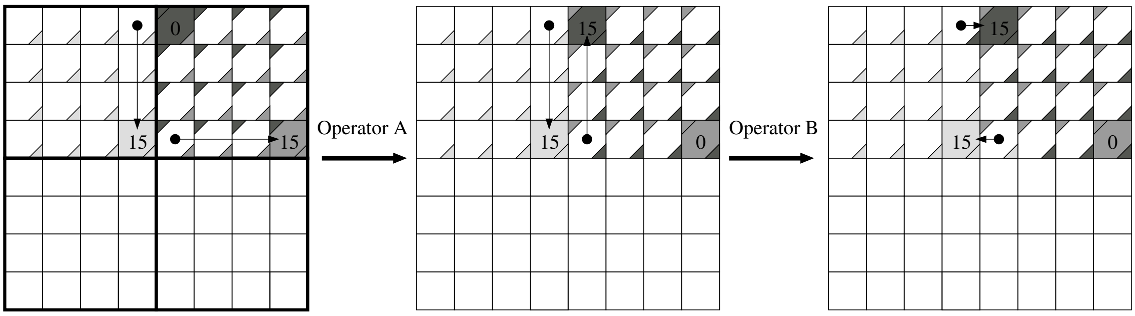


Figure 4.7: Application of a sequence of operators to the initial organization of the search, with a sharing degree of 16, to obtain an improved solution. The numbers represent LLC bank labels (0 and 15 in this example). The arrows show the LLC banks accessed by two cores for label 15 (3 links away in both cases). The color triangles provide further information about the effects of the operators. The color of the upper left corner triangle of a core matches the color of the LLC bank accessed for label 0. The lower right corner triangle does the same for label 15. The central figure shows the result of applying Operator A to two LLC banks, which interchange their labels (0 and 15). Any core that accessed one of these LLC banks for label 0 has to access the other LLC bank after applying the operator (notice the change of colors in the triangles and the related movement of the arrows). The figure on the right shows the result of applying Operator B to the two relevant cores in the example, which interchange their accessed LLC banks for bank label 15 (again, notice the change of colors in the triangles of those tiles and the related movements of the arrows). This results in a distance reduction of 2 links between each of these cores and their accessed LLC banks for bank label 15, which are now adjacent to the cores.

4.2.1 Exploring the DAPSCO Search Space

In order to search for optimal (or near-optimal) DAPSCO configurations for both meshes and tori, we have used two well-known global optimization algorithms: hill climbing [129] and simulated annealing [102]. The methodology explained in this section is applicable to any interconnection network topology. These heuristic algorithms start with the traditional partially shared LLC organization, in which clusters of cores share their LLC banks. This is a valid organization, as it trivially satisfies all the constraints of the graph problem stated before (it is the configuration that inspired these constraints). Then, the algorithms evaluate random valid organizations that originate by applying problem-dependent operators. For this particular problem, we have defined the following operators:

- Operator A: Two random LLC banks with different bank labels interchange their labels. In addition, the cores accessing each of these LLC banks are also interchanged —they *follow* the label— to maintain their whole memory space accessibility.

- Operator B: For a random bank label, two cores that access different LLC banks interchange these LLC banks.

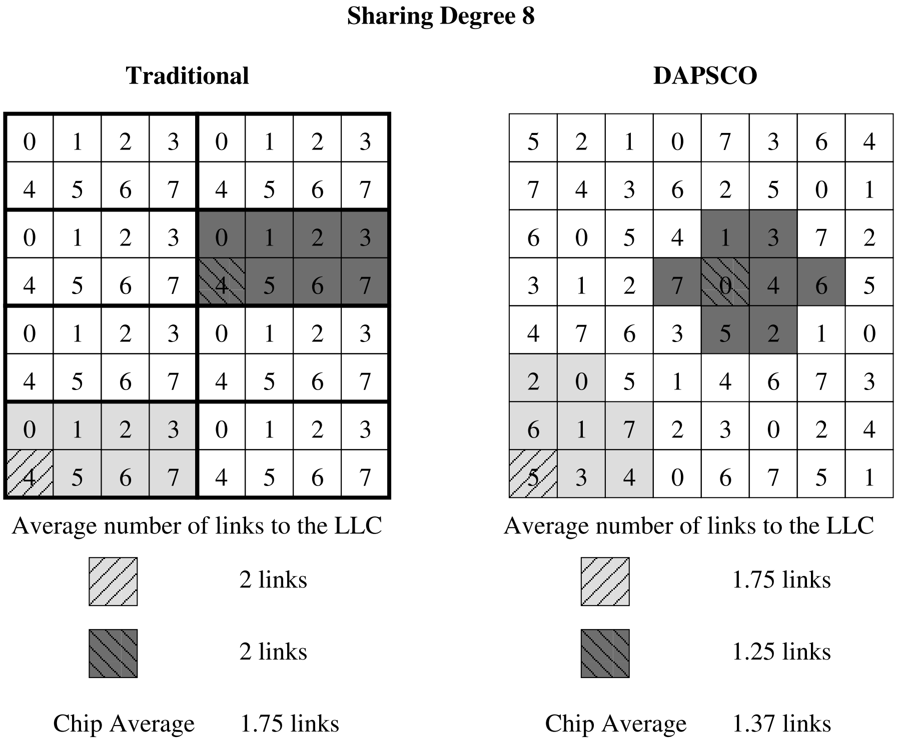
Figure 4.7 shows the application of the two operators on the initial cache organization, providing immediate improvements in the distance from the cores to the LLC. It is straightforward that applying any of these operators to a valid LLC organization produces another valid LLC organization. Furthermore, every valid organization can be reached from any other valid organization by applying an appropriate sequence of these two operators.

Figures 4.8, 4.9 and 4.10 compare the LLC access patterns of some cores in the traditional partially shared LLC organization (the one used as the initial state for our search algorithms), and in the best DAPSCO found by our algorithms, for a 64-core CMP with sharing degrees of 8, 16 and 32 in a mesh network. Notice that, for these sharing degrees, DAPSCO reduces the average distance to the LLC from 1.75 to 1.37, from 2.5 to 2.14 and from 3.88 to 3.35, respectively.

4.2.2 Tori and Sliding Patterns

In the case of tori, a simpler method can be used to obtain an optimal (or near optimal) partially shared cache organization. We call "access pattern" to the shape of the group of tiles whose banks are accessed by a core. Thanks to the symmetry of tori, the same access pattern can be used by every core to access the LCC banks provided that the access pattern can be used to tessellate the chip, as explained below in this section. We call these access patterns "sliding patterns".

In order to find an optimal sliding pattern we should start by finding all the access patterns with minimum total distance. For this, we start by choosing any tile as the central tile for the pattern. This central tile represents both the core that uses the pattern and one of the LLC banks accessed by the core. Then, the nearest tile to the center is added, representing another LLC bank accessed by the core in the central tile. Tiles are added following this policy until the pattern contains s tiles, where s is the sharing degree. Since several tiles may be at the same distance from the central tile, any one of them can be added at each step, resulting in different candidate access patterns. It is straightforward that the



Sharing Degree 8

Traditional

0	1	2	3	0	1	2	3
4	5	6	7	4	5	6	7
0	1	2	3	0	1	2	3
4	5	6	7	4	5	6	7
0	1	2	3	0	1	2	3
4	5	6	7	4	5	6	7
0	1	2	3	0	1	2	3
4	5	6	7	4	5	6	7

Average number of links to the LLC

▨ 2 links

▨ 2 links

Chip Average 1.75 links

DAPSCO

5	2	1	0	7	3	6	4
7	4	3	6	2	5	0	1
6	0	5	4	1	3	7	2
3	1	2	7	0	4	6	5
4	7	6	3	5	2	1	0
2	0	5	1	4	6	7	3
6	1	7	2	3	0	2	4
5	3	4	0	6	7	5	1

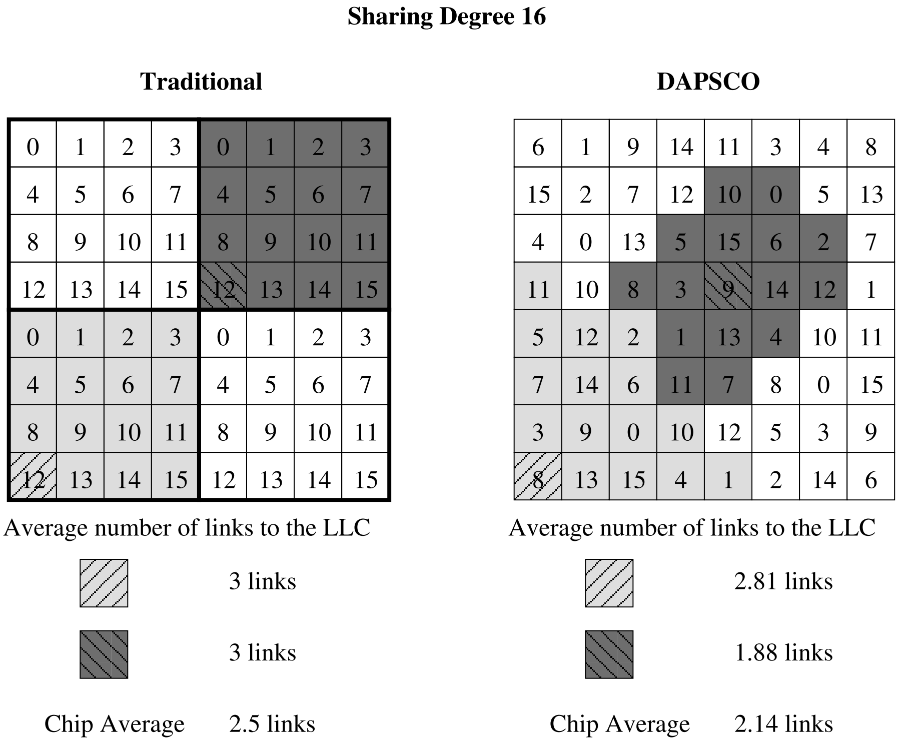
Average number of links to the LLC

▨ 1.75 links

▨ 1.25 links

Chip Average 1.37 links

Figure 4.8: LLC banks accessed by cores in a mesh. Sharing degree of 8. Traditional partially shared LLC organization (left) and DAPSCO (right). Each striped core accesses the shaded LLC banks surrounding it with the same background color. The numbers on the tiles represent the bank label. Notice that each core always accesses one LLC bank for each bank label, resulting in accessibility to the whole memory space. The average number of links traversed to reach the LLC banks is shown under the figures. DAPSCO significantly reduces this number of links.



Sharing Degree 16

Traditional

0	1	2	3	0	1	2	3
4	5	6	7	4	5	6	7
8	9	10	11	8	9	10	11
12	13	14	15	12	13	14	15
0	1	2	3	0	1	2	3
4	5	6	7	4	5	6	7
8	9	10	11	8	9	10	11
12	13	14	15	12	13	14	15

Average number of links to the LLC

▨ 3 links

▨ 3 links

Chip Average 2.5 links

DAPSCO

6	1	9	14	11	3	4	8
15	2	7	12	10	0	5	13
4	0	13	5	15	6	2	7
11	10	8	3	9	14	12	1
5	12	2	1	13	4	10	11
7	14	6	11	7	8	0	15
3	9	0	10	12	5	3	9
8	13	15	4	1	2	14	6

Average number of links to the LLC

▨ 2.81 links

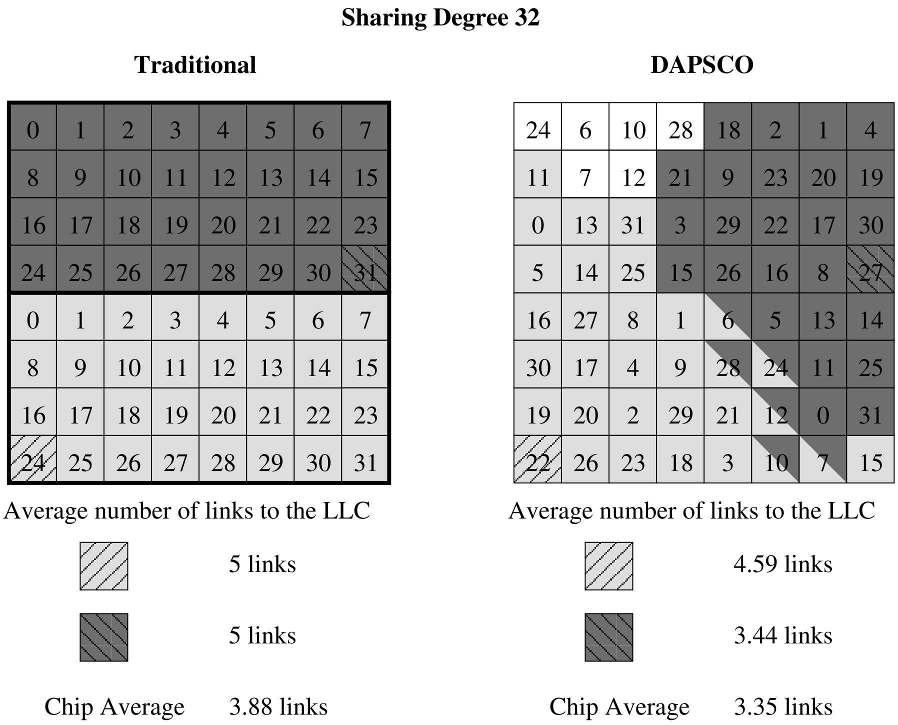
▨ 1.88 links

Chip Average 2.14 links

Figure 4.9: LLC banks accessed by cores in a mesh. Sharing degree of 16. Traditional partially shared LLC organization (left) and DAPSCO (right). Each striped core accesses the shaded LLC banks surrounding it with the same background color. The numbers on the tiles represent the bank label. Notice that each core always accesses one LLC bank for each bank label. The average number of links traversed to reach the LLC banks is shown under the figures. DAPSCO significantly reduces this number of links.

distance between the core and the LLC banks in all the patterns constructed by this method is the minimum possible.

As we said before, a cache organization must ensure that every core accesses one LLC bank for each bank label, so that the core has access to the whole memory space. Hence, we must assign bank labels to the tiles in such a way that the access pattern allows every core to access one LLC bank for each label. Unfortunately, this can be done only for patterns which can tessellate the chip (sliding patterns). In our case, a tessellation consists of dividing the chip in several polygons with the shape of the pattern. These polygons must not overlap and must cover all the tiles of the chip. When tessellating, the symmetric topology of the torus network must be taken into account, and a polygon that extends



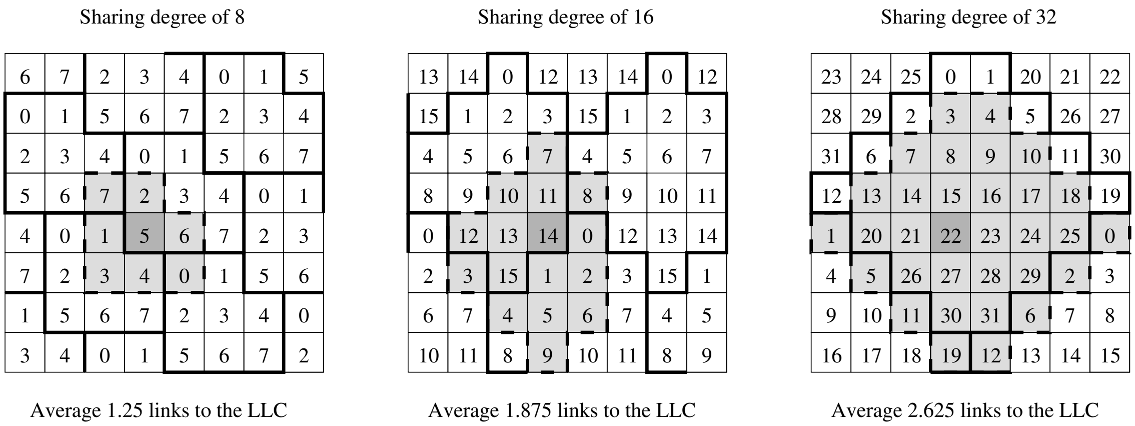
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

24	6	10	28	18	2	1	4
11	7	12	21	9	23	20	19
0	13	31	3	29	22	17	30
5	14	25	15	26	16	8	27
16	27	8	1	6	5	13	14
30	17	4	9	28	24	11	25
19	20	2	29	21	12	0	31
22	26	23	18	3	10	7	15

Figure 4.10: LLC banks accessed by cores in a mesh. Sharing degree of 32. Traditional partially shared LLC organization (left) and DAPSCO (right). Each striped core accesses the shaded LLC banks surrounding it with the same background color. The numbers on the tiles represent the bank label. Notice that each core always accesses one LLC bank for each bank label. The average number of links traversed to reach the LLC banks is shown under the figures. DAPSCO significantly reduces this number of links.

beyond one edge of the chip continues through the other end of the chip in the tessellation (that is, it wraps around). All the polygons must have the same orientation (that is, they cannot be flipped or rotated). If such a tessellation exists, the access pattern is a sliding pattern. The bank labels can be assigned to the tiles in any order as long as it is the same relative order for every polygon in the tessellation. This ensures that every core can access one LLC bank for each label by means of the sliding pattern.

Figure 4.11 shows the optimal sliding patterns and label assignations for a 64-tile CMP with a torus interconnection network for sharing degrees of 8, 16 and 32. Notice that no tessellation of a 64-tile chip exists with a minimum distance pattern for a sharing degree of 16 (we explored the full search space to check it),



Sharing degree of 8

6	7	2	3	4	0	1	5
0	1	5	6	7	2	3	4
2	3	4	0	1	5	6	7
5	6	7	2	3	4	0	1
4	0	1	5	6	7	2	3
7	2	3	4	0	1	5	6
1	5	6	7	2	3	4	0
3	4	0	1	5	6	7	2

Average 1.25 links to the LLC

Sharing degree of 16

13	14	0	12	13	14	0	12
15	1	2	3	15	1	2	3
4	5	6	7	4	5	6	7
8	9	10	11	8	9	10	11
0	12	13	14	0	12	13	14
2	3	15	1	2	3	15	1
6	7	4	5	6	7	4	5
10	11	8	9	10	11	8	9

Average 1.875 links to the LLC

Sharing degree of 32

23	24	25	0	1	20	21	22
28	29	2	3	4	5	26	27
31	6	7	8	9	10	11	30
12	13	14	15	16	17	18	19
1	20	21	22	23	24	25	0
4	5	26	27	28	29	2	3
9	10	11	30	31	6	7	8
16	17	18	19	12	13	14	15

Average 2.625 links to the LLC

Figure 4.11: Tessellations (solid lines) and label assignations to LLC banks for sharing degrees of 8, 16 and 32 in a 64-tile CMP. The dashed lines and the shaded tiles show the sliding pattern applied to the dark gray tile. The resulting average number of links to access the LLC in each configuration is shown. Compare these values with those of a traditional partially shared LLC with a torus: 1.75 links for a sharing degree of 8, 2.5 for a sharing degree of 16 and 3.25 for a sharing degree of 32.

and a sub-optimal sliding pattern with an additional link in the total distance is shown instead. When no optimal sliding pattern exists, the global optimization algorithms explained in Section 4.2.1 may find a better solution than the best possible sliding pattern. This is the case of a 64-tile CMP with a sharing degree of 16.

4.2.3 Implementation Details

Implementing DAPSCO requires just small changes in the hardware of traditional partially shared cache organizations. In the two-level cache hierarchy assumed in this chapter, three families of mapping functions used by the tiled CMP require modifications. Notice that these functions already exist in the circuitry of a CMP integrating a partially shared LLC; DAPSCO only needs different functions.

The first family of functions (family 1 from now on) performs the mapping of block addresses to L2 banks. Each core uses one function that, given the bank selector in the address of the block, chooses the predetermined L2 bank that is always accessed by that particular core for that bank selector upon an L1 cache miss or a write-back.

The second family of functions (family 2 from now on) performs the mapping of the bits in the sharing vector of the first level directory to identifiers of L1 banks. Each L2 bank uses one function to map each of the s bits of the sharing vector (s is the sharing degree) to one of the s L1 banks that can hold copies of blocks provided by that L2 bank.

The third family of functions (family 3 from now on) performs the mapping of the bits of the sharing vector of the second level directory to identifiers of L2 banks. Given the bank selector contained in the address of a block, there are n/s LLC banks with the corresponding bank label (n is the number of tiles in the CMP) which may contain a copy of the block. For each of the s bank labels, one function is necessary to perform this mapping.

These three families of mapping functions can be implemented in a number of ways. We have considered two options: using small tables or using small combinational circuits.

The simplest and most flexible way of implementing these functions is by using one small table for each one. For family 1, each core needs a table, indexed by bank label, which contains s entries of $log_2(n)$ bits. Each entry of the table represents the LLC bank that the core must access for a given bank label.

For family 2, each L2 bank needs a table that contains s entries of $log_2(n)$ bits. Each entry represents the L1 cache corresponding to a given bit in the sharing vectors.

For family 3, one table per bank label contains n/s entries of $log_2(n)$ bits. There are s of these tables, and they are accessed as follows: given a block address, the bank label corresponding to the block is used to choose the table to be accessed; given a bit of a sharing vector of the second level directory, the corresponding entry of the chosen table is looked up to get the LLC bank represented by the bit.

The contents of these tables are different in DAPSCO in order to use the new optimized mapping. Nevertheless, both the traditional partially shared organization and DAPSCO need exactly the same hardware with this implementation approach.

Alternatively, these tables can be optimized into combinational circuits so as to reduce their overhead, in the cases of both the traditional mapping and DAPSCO.

In order to calculate the overhead of DAPSCO using this approach, we have generated possible circuit layouts for the three families of functions. We have calculated these layouts for the baseline partially shared LLC and also for DAPSCO, in order to compare both organizations. We have found that, in the

Table 4.2: Characteristics of DAPSCO's address-to-bank circuits for a sharing degree of 8.

Number of cores	Average number of transistors per tile	No. of transistors in the longest critical path of any circuit
64	64	8
128	72	8
256	72	8
512	75	8

case of the circuits that translate sharing codes into cache bank identifiers (i.e., families 2 and 3), there are no noticeable differences between fixed-boundary clusters and DAPSCO in neither the total number of transistors nor the number of transistors in the critical path of these circuits, regardless of the number of cores or the sharing degree. Therefore, no overhead is introduced by DAPSCO due to its different implementation of the circuits of families 2 and 3.

However, the address-to-bank mapping circuits (family 1), is indeed more complex in DAPSCO. This address-to-bank mapping is straightforward in the baseline partially shared LLC, since the bank label bits of the block address can be used to generate the identifier of the L2 bank to access (as shown in Figure 4.4), while DAPSCO needs a combinational circuit to perform this task. In general, the size of this circuit is small. As an example, Table 4.2 shows the number of transistors needed by the address-to-bank map circuits of DAPSCO for varying numbers of cores, when a mesh and a sharing degree of 8 are used. This table shows the average number of transistors per tile and the maximum length of the critical path of any of these circuits in the chip. Fortunately, these additional transistors are negligible in terms of power and area. In addition, the overhead of this circuit scales gracefully with the number of cores, and the maximum critical path remains constant for any number of cores tested. As for latency, this circuit is easily traversed in one cycle and can be accessed in parallel to the L1 cache tags. Therefore, no latency is added to the critical path of cache misses.

As a conclusion, no noticeable increase in power, area or latency is produced by any of the modifications of the circuitry needed by DAPSCO.

4.3 Evaluation

4.3.1 Effectiveness of DAPSCO

In order to search for the best possible DAPSCO, we used both a hill-climbing algorithm and a simulated annealing algorithm, as noted in Section 4.2.1. These algorithms were executed repeatedly for several combinations of sharing degree and number of cores, and we selected the best solution that they found for each combination. In total, the instances of these algorithms ran in parallel on machines equipped with 2.33GHz Intel Xeon processors for approximately one week. Each instance was restarted every time that the consecutive generation of two million LLC organizations did not improve the best organization found in the current execution. We tested optimizations such as only applying operators that improved the average distance to the LLC from one of the cores affected by the operators in order to direct the search.

Figure 4.12 shows the average number of links traversed for the cores to reach the LLC, in chips containing from 64 to 512 cores with a sharing degree ranging from 4 to 256. These results are shown for meshes and tori. Three different values are presented for each combination of network topology, core count and sharing degree: the base mapping, an optimistic bound and the best DAPSCO found by the algorithms. The optimistic bound assumes that every core accesses its closest LLC banks. This is not always possible in practice, but we include it as an upper bound to check the effectiveness of the heuristics employed. In fact, no real DAPSCO configuration can match the optimistic bound in a mesh for sharing degrees of four or more. Intuitively, this is caused by the impossibility of every core to access its closest LLC banks in a correct configuration. If every core accessed its closest LLC banks, the banks in the corners (and borders) of the chip would be accessed by fewer cores than those banks in the center of the chip, resulting in an invalid configuration. As the sharing degree approaches the number of cores in the chip, this inefficiency becomes more noticeable for the mesh.

The most important result is that, as the sharing degree increases, the number of links traversed to access the LLC grows too in all configurations, and at the same time the percentage of traversed links avoided by DAPSCO also increases. Hence, our proposal will have more beneficial effects in the performance of the system for higher sharing degrees. For instance, at 512 cores and a sharing degree of 128 on a torus, the latency (and energy consumption) required to reach the LLC can potentially improve by almost 50%. In addition, as the sharing degree

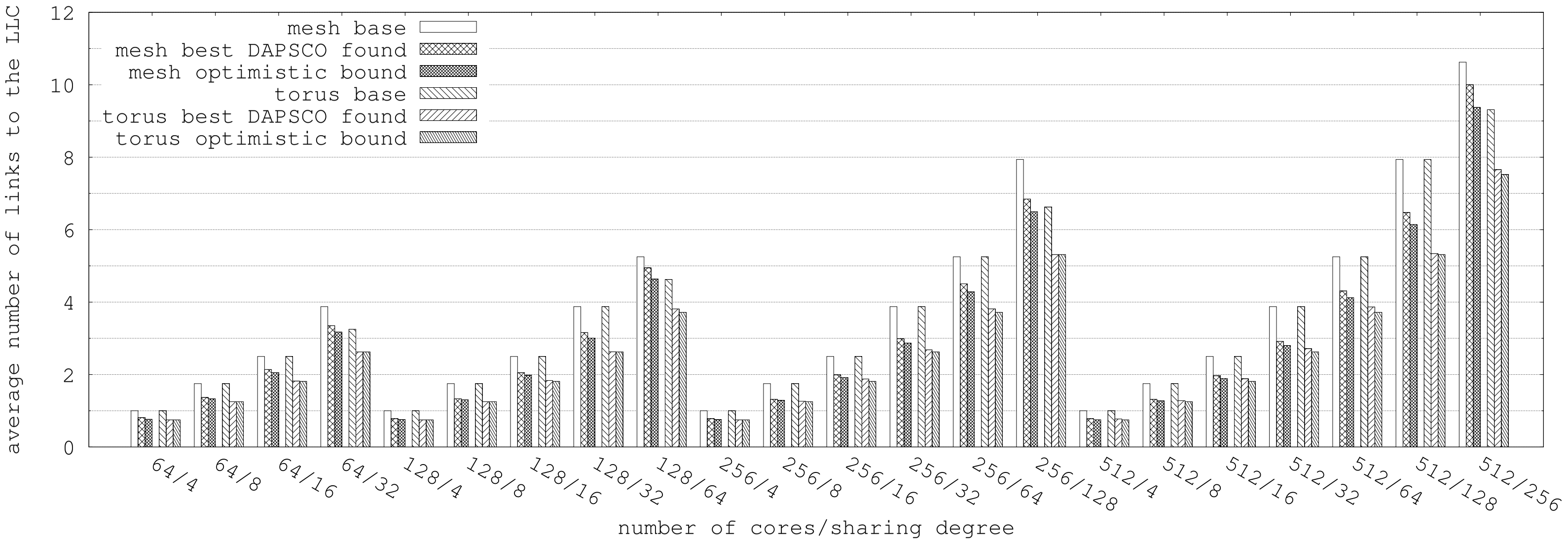


Figure 4.12: Average number of links from cores to LLC banks. X-axis labels represent the number of cores in the CMP (from 64 to 512) and the sharing degree (from 4 to 256).

Table 4.3: System characteristics.

Processors	64 UltraSPARC-III+ 3 GHz. 2-ways, in-order.
L1 Cache	Split I&D. Size: 64KB. Associativity: 4-ways. 64 bytes/block. Access latency: 1 cycle.
L2 Cache	Size: 1MB each bank. 64MB total. Associativity: 8-ways. 64 bytes/block. Access latency: 2 (tag) + 3 (data) cycles.
RAM	4 GB DRAM. 8 memory controllers along the edges of the chip. Memory latency: 150 cycles + on-chip delay. Page Size: 4 KB.
Interconnection	Bi-dimensional mesh and torus 8x8. 16 byte links. Latency mesh: 2 cycles/link. Latency torus: 4 cycles/link. (in absence of contention) Flit Size: 16 bytes. Control packet size: 1 flit. Data packet size: 6 flits.

increases, fixed latencies (such as the accesses to the tag and data arrays of the caches) account for a smaller fraction of the total time to resolve each cache miss, while the latency to traverse links and routers takes a larger fraction, making DAPSCO improvements more significant in practice.

4.3.2 Simulation Methodology

We use the GEMS [126] simulator to model a tiled-CMP whose characteristics can be seen in Table 4.3. We use the workloads described in Table 4.4. For virtualized workloads we use Virtual-GEMS [56]. We model a detailed interconnection network with the Garnet [5] network simulator. Energy consumption figures for the network are obtained from the Orion 2.0 [94] power model.

Our goal is to compare the traditional partially shared cache organization described in Section 4.1.1 and DAPSCO. We have limited our evaluation to 64-tile

Table 4.4: Benchmark configurations.

Workload	Description	Size
lu64p	Factorization of a dense matrix	512x512 matrix
lunc64p	Factorization of a dense matrix, non-contiguous memory	512x512 matrix
tomcatv64p	Vectorized mesh generation	256
volrend64p	Ray-casting rendering	Head
watersp64p	Optimized molecular dynamic simulation of water	512 molecules
apache4x16p	Virtualized web server with static contents	Four 16-core VMs with 500 clients each, 10ms between requests of a client
jbb4x16p	Virtualized Java server	Four 16-core VMs with 1.5 warehouses per core

CMPs due to simulation time constraints, despite the fact that DAPSCO would obtain more favorable results with more tiles. In total, we have evaluated four different LLC configurations:

- **8Traditional**: traditional mapping for partially shared caches in which every 8 tiles make up a cluster and share their L2 banks.
- **8DAPSCO**: DAPSCO with a sharing degree equal to 8.
- **16Traditional**: traditional mapping for partially shared caches in which every 16 tiles make up a cluster and share their L2 banks.
- **16DAPSCO**: DAPSCO with a sharing degree equal to 16.

Two different network topologies, mesh and torus, were used in the tests. Torus links are longer than mesh links, and therefore we set their latency to twice that of mesh links. The suffixes Mesh or Torus are added to the names of the configurations in order to identify the network topology used (e.g., 8TraditionalTorus for the traditional configuration with a sharing degree of 8 using a torus network.)

Table 4.5 shows the average number of links traversed by messages from arbitrary cores to arbitrary banks of the LLC in each configuration. The rest of the machine remains the same for all the configurations (Table 4.3).

Table 4.5: Average distance from cores to LLC banks.

Configuration	Average distance to the LLC (links)	Improvement over traditional
8TraditionalMesh	1.75	
8DAPSCOMesh	1.37	24%
16TraditionalMesh	2.5	
16DAPSCOMesh	2.14	17%
8TraditionalTorus	1.75	
8DAPSCOTorus	1.25	40%
16TraditionalTorus	2.5	
16DAPSCOTorus	1.88	33%

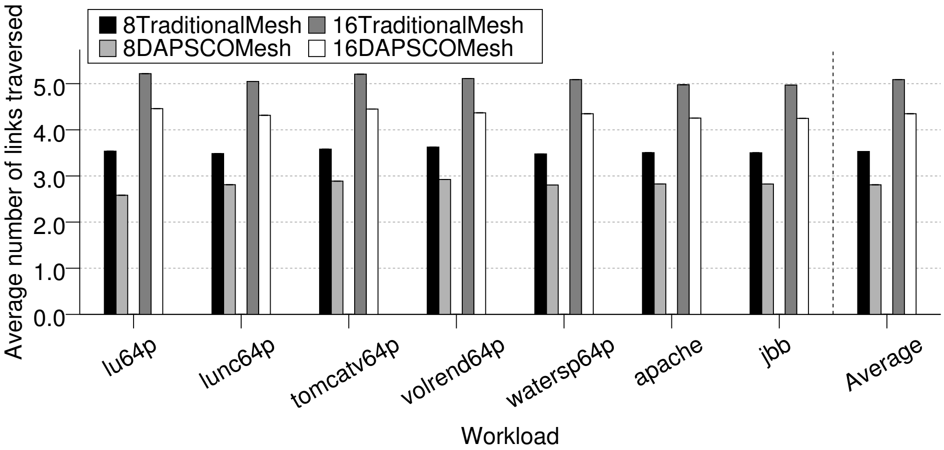


Figure 4.13: Average number of links traversed to resolve 2-hop cache misses using a mesh network.

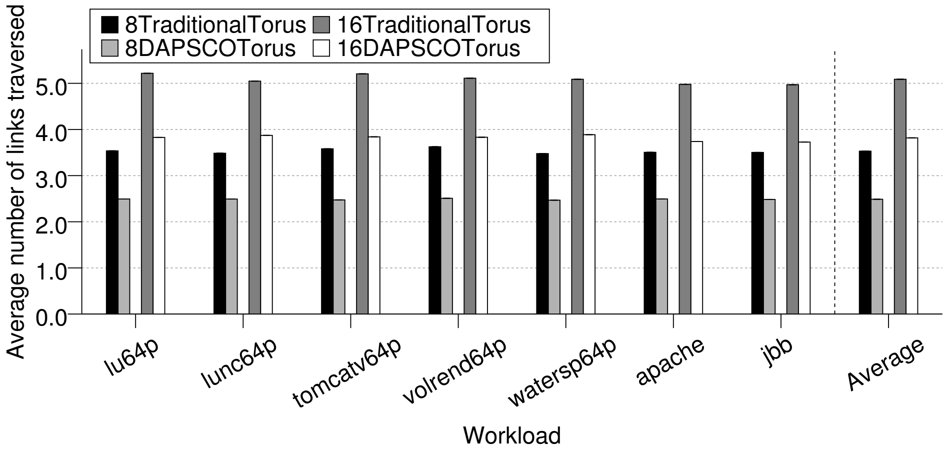


Figure 4.14: Average number of links traversed to resolve 2-hop cache misses using a torus network.

4.3.3 Results and Discussion

Our results confirm that in partially shared LLCs, most L1 cache misses are resolved with just two hops to retrieve the data from the L2 cache, as we pointed out in Section 4.1.3. These misses benefit the most from DAPSCO in terms of traversed links, since both hops get advantage from the reduced distance to the L2 bank accessed. Figures 4.13 and 4.14 show the average number of links traversed to resolve these misses in the mesh and torus networks, respectively. These experimental results for 2-hop misses with the torus (improvements of 40% and 33% for sharing degrees of 8 and 16) and the mesh (improvements of 24% and 17% for sharing degrees of 8 and 16) match almost perfectly the values previously calculated for the average number of links traversed to access the LLC showed in Table 4.5. Other kinds of misses also experience a reduction in the number of traversed interconnection network links, although smaller.

Since the 2-hop miss kind is the most frequent one and it gets full advantage of DAPSCO in all the hops of its critical path, DAPSCO has a noticeable impact on the overall latency and energy consumption of cache misses.

In the end, the reduction in links traversed to resolve misses translates directly into reductions in both execution time and network energy consumption. Figures 4.15 and 4.16 show the execution time of the eight configurations tested. In the case of the mesh (Figure 4.15), the performance of the system improves by 4% and 6% when using DAPSCO with sharing degrees of 8 and 16, respectively. In the case of the torus (Figure 4.16), the performance improvement of DAPSCO

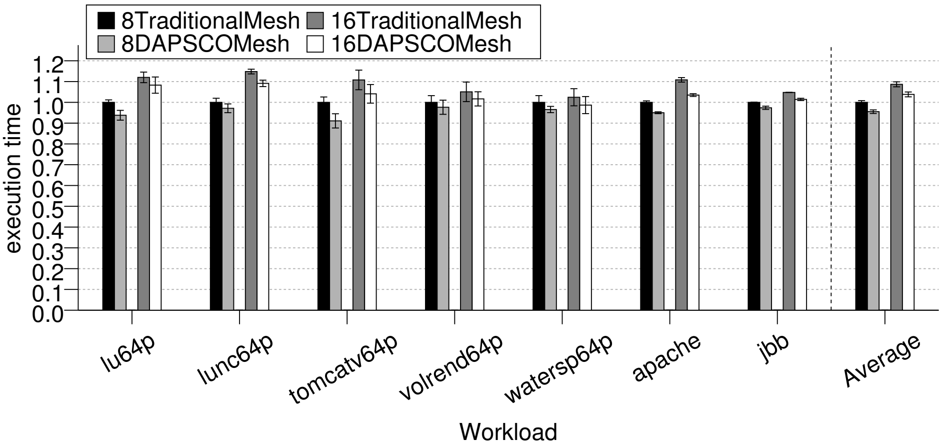


Figure 4.15: Normalized execution time using a mesh interconnection network.

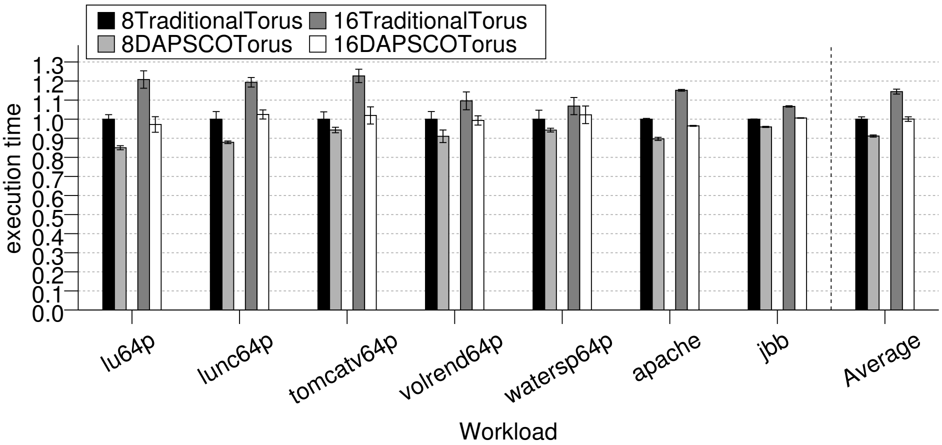


Figure 4.16: Normalized execution time using a torus interconnection network.

rises to 10% and 13% for sharing degrees of 8 and 16. We can see that despite the average reduction in traversed links being smaller in the case of a sharing degree of 16, the gain in execution time is slightly higher than for a sharing degree of 8. The cause for this was mentioned previously: the distance traversed to resolve cache misses is higher for larger sharing degrees. Hence, the percentage of cache miss latency due to link and router traversals increases in comparison with fixed latencies (e.g., cache tag and data array accesses), and this makes the benefits of DAPSCO more noticeable for a larger sharing degree.

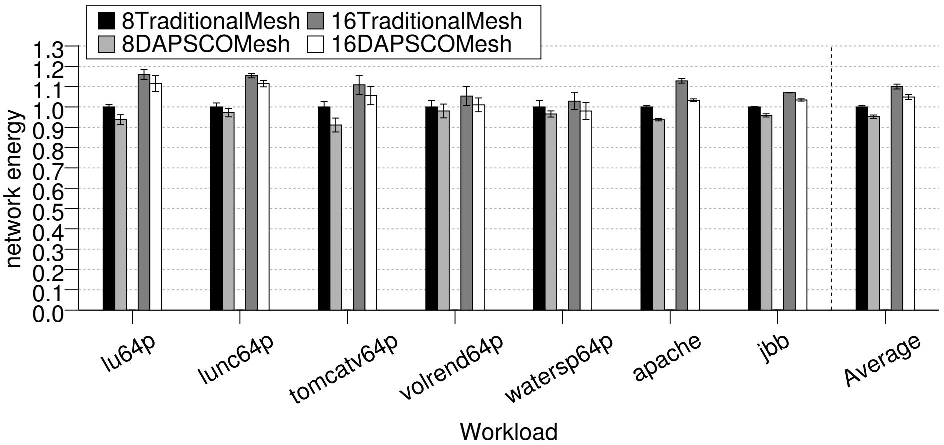


Figure 4.17: Normalized energy consumption of the mesh interconnection network.

Figures 4.17 and 4.18 show the normalized energy consumption of the network for the torus and the mesh, respectively. Again, thanks to the reduction in links and routers traversed to resolve misses, energy consumption gets reduced when using DAPSCO by 4% and 6% with a mesh, and by 10% and 13% with a torus for sharing degrees of 8 and 16, respectively. There is a clear parallelism between the results of performance and network energy for every benchmark.

All of these results show the effectiveness of DAPSCO. In fact, in the worst benchmark for DAPSCO, it still raises the performance of the system by 2% with a mesh and by 5% with a torus.

However, the full potential of DAPSCO is not reached with the tested configurations. As the number of cores and the sharing degree of the partially shared caches grow, the benefits of DAPSCO will become more noticeable, since both the percentage of links removed from the path to the LLC banks and the percentage of the execution time due to link traversals increase.

As for the performance comparison between sharing degrees, which is not the goal of this chapter, a sharing degree of 8 performs better than a sharing degree of 16 in our tests, but we believe that this could change with a wider set of workloads with larger working sets, as proven by other studies.

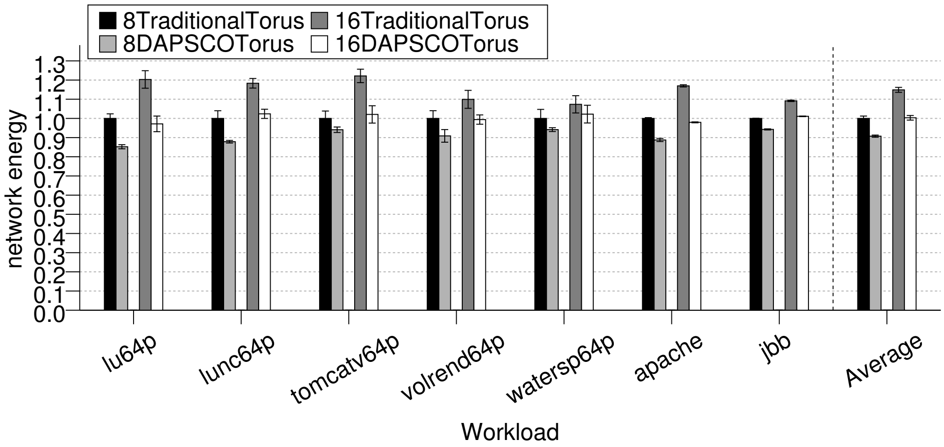


Figure 4.18: Normalized energy consumption of the torus interconnection network.

4.3.4 Orthogonality of DAPSCO: Reactive NUCA

DAPSCO is orthogonal to other proposals that are based on traditional partially shared caches. In this section, we prove it by applying DAPSCO to one of these proposals, known as Reactive NUCA [76]. Reactive NUCA is based on dynamically classifying the blocks accessed by the cores in several pre-determined types, and then applying the best pre-determined sharing degree to each of these types to improve performance. Hardavellas et al. [76] considered three different types to which the blocks can belong: instructions, private data and shared data. They determined that the sharing degrees that optimized performance for these types in a 16-core CMP with a torus network were: four for instructions, one for private data (equivalent to using private caches for private data) and sixteen for shared data (equivalent to using a shared cache that avoids block replication for shared data).

We have applied Reactive NUCA to a 64-core CMP with a mesh network. When a mesh is used, Reactive NUCA uses traditional partially shared caches for the LLCs. We have considered the same block types as the original proposal of Reactive NUCA (instructions, private data and shared data), and by performing exhaustive tests, we determined that the sharing degrees for these types of data that yield the best performance in our benchmarks are four for instructions, one for private data and eight for shared data. Then, we have modified Reactive NUCA to replace the mapping of the partially shared caches with DAPSCO. Figure 4.19 shows the performance comparison between Reactive NUCA and

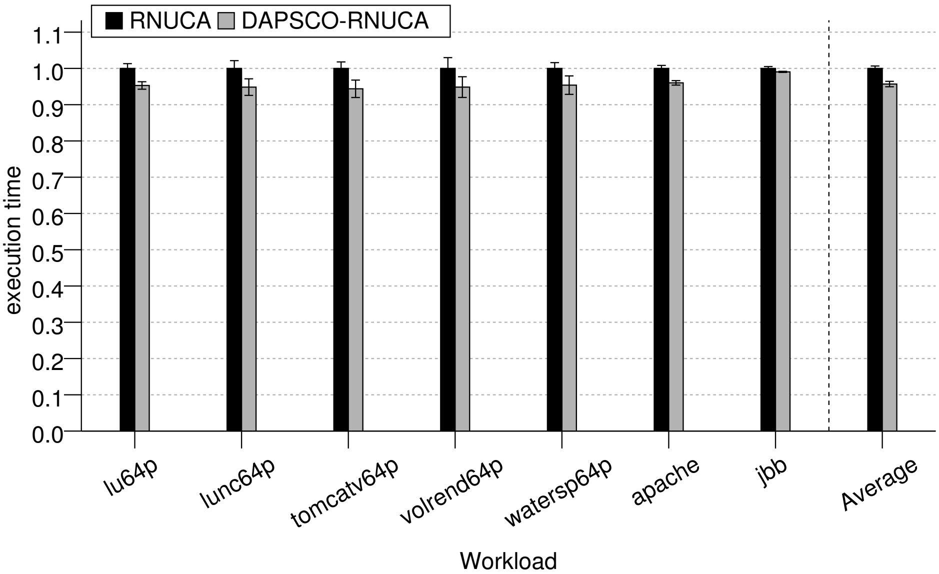


Figure 4.19: Execution time of Reactive NUCA without and with DAPSCO.

Reactive NUCA plus DAPSCO. These results show that the performance of Reactive NUCA improves by 3% in average (and up to 7% in some benchmarks) when DAPSCO is enabled, thanks to the reduction in the links traversed to access the LLC banks provided by DAPSCO, which again matches the theoretical values. Hence, DAPSCO can be beneficial when combined with optimized LLC organizations, as well as with the traditional LLC organization discussed previously.

4.4 Related Work

Several works study the effects of using partially shared caches of different sizes. Huh et al. [87] test five different sharing degrees for a 16-core CMP, from a completely private LLC to a completely shared LLC, by using a traditional block mapping for partially shared caches. Their results show that the sharing degree that achieves the best performance depends on the workload executed. DAPSCO would improve the performance and energy consumption of the system in all the partially shared configurations tested in their work.

Hammoud et al. [75] show that different sharing degrees for the LLC are more appropriate for different execution phases of the same application. They developed a technique for dynamically varying the sharing degree of the LLC based on real-time feedback on the behavior of the applications. DAPSCO could be used along with this technique, instead of the traditional block mapping, to further improve the performance and energy consumption of the system.

Dynamic (or adaptive) mappings provide more flexibility by not fixing cache banks where blocks are mapped, which can be used to improve performance by dynamically bringing blocks to banks closer to the requesting cores. However, complex and power-consuming lookup mechanisms, as well as extra structures, are needed to locate blocks in this case, and the cache coherence protocol may become noticeably more complex. On the other hand, static mappings like DAPSCO, which map each memory block to a fixed cache bank, are simpler to implement than dynamic mappings, require less area, and have been traditionally used in commercial machines [105, 161].

In particular, regarding cache miss latency in CMPs, many proposals [16, 33, 76, 100] reduce it by dynamically trying to allocate copies of the block as close as possible to the requestors, but these techniques commonly increase network traffic and need power-consuming lookup mechanisms to locate blocks. In contrast, DAPSCO consumes less energy and reduces execution times as a result of an increase in data proximity, requiring almost no extra hardware and without complicating the design of the chip.

Reactive NUCA [76] introduces the concept of fixed-center clusters which, similarly to the sliding patterns described as a part of DAPSCO, make each core access a different subset of the L2 banks to replicate data (without increasing the capacity pressure of the cache) and enable fast nearest-neighbor communication. However, fixed-center clusters only work for torus networks, which can be seen as an important shortcoming, since current CMPs proposals commonly use mesh networks which are easier and cheaper to implement and deploy. Additionally, the rotational interleaving that is used to create these fixed-center clusters produces sub-optimal distances to the LLC and requires that cluster size be a power of two and smaller than half the number of cache banks of the CMP, limiting its flexibility and effectiveness compared to DAPSCO, which suffers none of these inefficiencies. For instance, rotational interleaving does not work for 32-bank clusters in a 64-core CMP. In fact, DAPSCO can be adapted to Reactive NUCA to further improve its energy consumption and performance in large CMPs, especially to improve the use of meshes in Reactive-NUCA, as shown in Section 4.3.4.

The operating system can take into account the address-to-cache-bank mapping when performing the virtual to physical address translation in order to map memory pages to certain cache banks so as to improve the performance of the system [36]. Distance-Aware Round-Robin Mapping [156] improves the mapping of the memory pages to cache banks in large NUCA caches by means of an OS-managed mechanism that at once reduces the distance from the data to the requestors and provides a fair utilization of the NUCA banks.

Plenty of research on reducing energy consumption in CMPs is being carried out. Regarding NoCs, cache coherence protocols can take advantage of heterogeneous networks to reduce power consumption by transmitting critical, short messages through fast power-consuming wires and non-critical messages through slower low-power wires [54]. As for cache architecture, TurboTag [117] uses bloom filters to avoid unnecessary tag lookups and reduce power consumption. However, all these proposals reduce energy consumption at the cost of degrading performance, while DAPSCO improves it.

Finally, a new tag directory buffer technique has been recently proposed as a good energy-delay trade-off between private and shared cache organizations [86]. Again, DAPSCO is orthogonal to this proposal.

4.5 Conclusions

We have proposed DAPSCO to optimize the cache organization of tiled CMPs by making each core access the LLC banks surrounding it on any network topology, minimizing the average number of network links traversed to access the LLC. Two heuristic algorithms have been used to explore the search space of the NP-complete problem of finding optimal DAPSCO mappings, applying them to CMPs integrating mesh and torus network topologies. An alternative method for constructing optimal sliding patterns for torus-based CMPs has been presented too.

The cost of DAPSCO is negligible in terms of hardware, and it achieves significant improvements in both the execution time of the system and the energy consumption of the interconnection network when compared to the traditional partially shared cache organization in which clusters of the tiles share their LLC banks.

We have shown two examples of DAPSCO that improve the performance of a 64-core CMP by 4% and 6% with an underlying mesh topology, and by 10% and 13% with an underlying torus topology, all of it with respect to traditional partially shared caches with sharing degrees of 8 and 16. Network power consumption also gets reduced by 4% and 6% (mesh), and by 10% and 13% (torus) regarding the same traditional configurations. We have also shown that as the number of cores and sharing degree increase, link traversals account for a growing fraction of execution time and energy consumption, and DAPSCO removes a higher percentage of links from the critical path of cache misses, becoming even more effective.

CHAPTER 5

In-Cache Coherence Information

In this chapter, we introduce In-Cache Coherence Information (ICCI), a new cache organization that leverages shared cache resources and flat coherence protocols to provide inexpensive hardware cache coherence for large core counts (e.g., 512), achieving execution times close to a non-scalable sparse directory while noticeably reducing the energy consumption of the memory system. Very simple changes in the system with respect to traditional bit-vector directories are enough to implement ICCI. Moreover, ICCI does not introduce any storage overhead with respect to a broadcast-based protocol, yet it provides large storage space for coherence information. ICCI makes smarter use of cache resources by dynamically allowing last-level cache entries to store either blocks or sharing codes. This way, just the minimum required number of directory entries are allocated. Besides, ICCI suffers only a negligible amount of directory-induced invalidations.

Results for a 512-core CMP show that ICCI reduces the energy consumption of the memory system by up to 48% compared to a tag-embedded directory, and up to 8% compared to the state-of-the-art Scalable Coherence Directory (SCD), which ICCI also outperforms in execution time. In addition, ICCI can be used in combination with elaborated sharing codes to apply it to extremely large core counts. We also show analytically that ICCI's dynamic allocation of entries makes it a suitable candidate to store coherence information efficiently for very large core counts (e.g., over 200 K cores), based on the observation that data sharing makes fewer directory entries necessary per core as core count increases.

5.1 Background

Cache coherence enables simple shared-memory programming models in the presence of private caches, facilitating the development of efficient parallel applications. As discussed in the introduction of this thesis, during the next years, hardware coherence will remain desirable for developing new non-structured parallel programs, as well as for running legacy applications [125]. Chips designed for market segments ranging from high-performance computing to embedded systems to cloud computing will benefit from a coherent shared-memory model. In addition, multi-cores are a more scalable design than single-cores of increasing complexity, as they can potentially scale out with linear area and power requirements with respect to core count. To achieve good scalability for multi-cores, these future chips need to integrate a scalable hardware cache coherence mechanism in terms of both area and energy consumption. A recent commercial example of a very large cache coherent system is the new SGI® UV2 [168] machine developed by Silicon Graphics.

In a chip multiprocessor, the tag arrays of the caches store address tags that identify what blocks are stored in the corresponding data blocks of the caches and control bits encoding the access permissions associated to each block. Any action required to maintain coherence, such as providing a valid copy of a block to a requestor or invalidating all sharers upon a write request, can be carried out based exclusively on the information stored in the tag arrays of the private caches. When scaling out CMPs to large core counts, the overhead of tag arrays remains constant per tile, insensitive to the number of cores. This is an apparently good starting point to develop cache coherence mechanisms that keep up with the desirable linear scalability properties interrelating area and power with speed expected from multi-cores. However, the particular characteristics of cache coherence make it very difficult to implement cache coherence maintaining all these linear properties in practice, as we will see next.

Snoopy-based coherence mechanisms [65] leverage tag information to maintain cache coherence, without requiring any additional sharing information to be stored. Every time that a memory request cannot be resolved in a node's private cache (i.e., the block address is not found in the tag array or the required access permissions are missing), the request is broadcast to all tiles. The appropriate coherence actions are carried out collectively by all nodes based on their private-cache tag information. Typically, just one or some of the nodes are required to take steps to resolve the request, be it by providing the block (on a read request) or by invalidating their private copies of the block (on a write request). Unfortunately, snoopy-based cache coherence protocols cannot scale beyond a few cores due to their broadcast-induced large volumes of network traffic and high number of energy-consuming cache lookups required to determine the actions to take on each memory request, based on the private-cache tag information distributed across all the tiles of the chip [97].

To mitigate some of the problems of snoopy-based coherence, a directory can be used to store all the information necessary to carry out any coherence transaction, which is the base of directory-based cache coherence protocols [12, 31, 69]. For instance, duplicate-tag directories [12] keep a copy of all the tags of the private caches, organized in a proper way to improve lookups. The duplicates of all the tags in the chip that belong to the same private-cache set number (i.e., all the tags that may refer to the same memory block) are stored physically together. On a memory request, a single network message reaching the appropriate storage point of the duplicate-tag directory allows the lookup of all the tags necessary to determine any required coherence actions. This removes the network problems of snoopy-based protocols, because it can be implemented with efficient point-to-point coherency traffic over scalable networks, such as meshes or tori, at the expense of requiring extra memory for storing the sharing information (i.e., the duplicated tags). Nevertheless, the per-core storage overhead of duplicate-tag directories is constant with respect to the number of cores, and therefore, they are scalable in terms of area. Unfortunately, the energy required to perform sharing-information lookups grows linearly with the number of cores (like in snoopy-based protocols), because a lookup requires an associative search with a degree of associativity proportional to the number of cores, to check the appropriate copies of the tags of all cores. This makes duplicate-tag directories impractical for large core counts for energy-efficiency reasons.

In general, address-indexed arrays of tags (such as the ones used by snoopy-based coherence and duplicate-tag directories) are an information storage scheme unable to facilitate efficient coherence information search. The underlying barrier to the scalability of cache coherence is that, despite the beneficial fact that the amount of tracked private-cache blocks per core is insensitive to core count, the total number of tracked blocks in the chip is not, and it grows proportionally to the number of cores. The coherence scheme must find, among all the private-cache block information in the chip, the specific pieces of information necessary to determine and carry out the particular coherence actions required by each memory request. In snoopy-based coherence and duplicate-tag directories, the amount of information checked for each memory request grows proportionally to the number of cores (and to the total amount of information in the chip), making the energy costs of maintaining cache coherence unaffordable for large core counts.

To overcome this energy barrier, coherence information must be stored in a way that, combined with an appropriate access mechanism, results in a natural filter that fetches the necessary information for coherence actions with low energy costs (and low latency) regardless of core count. Doing this requires more complex structures than tag arrays, also with larger storage footprints. This complicates the task of using structures that take as little space as possible, as we would like their per-core storage overhead to be insensitive to core count (that is, like tag arrays).

In summary, computer architects must design coherence schemes that at the same time are fast, take little space, and spend little energy for each lookup. All of it should be achieved regardless of core count if these schemes are to be scalable for large CMPs. Unfortunately, designing such a scalable cache coherence scheme comprising all these properties is very problematic in practice, and as chip multiprocessors scale out to large numbers of cores (e.g., hundreds), cache coherence becomes a serious barrier affecting area, energy and performance very negatively.

In an attempt to achieve an approximation to all these properties, large systems usually rely on directory-based cache coherence protocols that encode sharing information in bit-vectors [31]. The directory is address-indexed, like duplicate-tag directories. However, instead of storing sets of tags as in duplicate-tag directories (in which several tags can be the same, representing different sharers), each directory entry stores one address tag, a few control bits, and a bit-vector containing the sharers of the block identified by the tag (with just one entry per tag). This enables efficient searches: given the address of a block, the bit-vector of sharers is returned after performing a fixed number of tag comparisons (the associativity of the directory) that does not depend on core count. A block for which no entry exists in the directory is not cached in private caches. In comparison, the number of tag comparisons in duplicate-tag directories was proportional to core count, limiting their scalability. Bit-vector directories provide a relatively good trade-off between the amount of information to check out in each memory request and the amount of storage used to optimize lookup energy. These directories are stored in set-associative arrays with a limited number of entries that suffer from address conflicts. When a directory entry is evicted from the set-associative cache due to a conflict, the blocks tracked by the evicted entry must be invalidated in order to prevent incoherences in the future caused by non-tracked copies of memory blocks. These invalidations are known as *directory-induced invalidations*.

The directory organization has to deal with two extreme cases that may arise during the execution of programs. These are the following:

a) All blocks in the private caches are private to the cores (no additional sharers exist for any of the blocks). This determines the maximum number of entries that will ever be used in the directory at once, one per private-cache entry. We define the *coverage* of a directory as the percentage ratio of the number of directory entries to the number of private-cache entries. A directory with fewer entries than those necessary for the worst case (i.e., with a coverage under 100%) is *underprovisioned*, and will eventually be unable to store all the coherence information for the particular applications running, limiting the effective capacity of the private caches and harming performance.

b) A block is shared by all cores. This determines the size of each entry, which must have enough room to store all the sharers of the block (that is, all cores). Usually, a full-map bit-vector is used, containing one bit to indicate the presence (or absence) of a copy of the block in each of the private caches.

Designing a directory able to handle both extreme cases typically results in a directory with both a high number of entries and a large size for each entry. As the number of nodes increases, both the total number of entries in the chip and their size (one bit per core) grow linearly, with multiplying effects on their area overhead. In general, this scheme becomes eventually unaffordable, as the per-tile overhead of the directory grows linearly with the number of cores.

To make things worse, the set-associative arrays used to store the directory suffer from address conflicts. To reduce the amount of conflicts, the directory must be *overprovisioned* (with a coverage over 100%), containing many more entries than strictly necessary for the worst case scenario, in order to avoid high amounts of directory-induced invalidations that would degrade performance notably.

Halting the growth of any of the two previously described elements (i.e., directory entry size or entry count), preferably without harming performance as a side-effect, would make the directory scalable, since its per-tile overhead would be constant with respect to core count. For this reason, the two classic ways to improve directory storage scalability are precisely reducing the size of directory entries and reducing the number of directory entries.

Entry size is commonly reduced by using more scalable sharing codes in the directory, instead of full-map bit-vectors (e.g., lists made of pointers [92] or trees [137] with logarithmic scaling properties, or hierarchical bit-vectors [125,159] with square-root scaling properties). These sharing codes are often *composable* in the sense that several directory entries can make up the sharing information regarding a single memory block. Elaborate codifications such as hierarchical protocols or trees result in complex coherence protocols to manage the distributed sharing codes, difficult to validate and safely deploy in chips. They also introduce other inefficiencies that have traditionally limited their applicability in practice, such as slow invalidations (e.g., invalidating the sharers in a list, one after another, like in SCI [92]).

Using inexact sharing codes [2,4,69,110,180,186] is another way to reduce the area footprint of directories. However, although they are scalable in area, the scalability in performance of these alternatives is in doubt. Code inexactitude causes many spurious coherence actions, with increasing occurrence as the disparity between core count and entry size grows (e.g., if we keep the same directory entry size while the core count rises). These spurious actions increase the amount of network traffic, the number of cache lookups, and as a result, they raise energy consumption and execution time. In addition, those inexact codes with the smallest overheads (e.g., limited pointers with logarithmic overhead) suffer from superfluous coherence actions the most [125].

On the other hand, the high number of required entries has been tackled by means of more efficient ways to hash the directory arrays [51,159] (preventing conflicts and enabling low overprovisioning in the directory), or by means of techniques that deactivate fine grain coherence tracking for some memory blocks (reducing the number of used entries in the directory [39,113], and likewise, allowing directories to suffer few conflicts). Still, directory coverages cannot go under 100%, which implies that the amount of directory entries in the chip keeps growing linearly with core count. Reducing the coverage under 100% to improve scalability would be very risky, as none of these techniques can safely ensure that the resulting underprovisioned directory (with fewer entries than there are tracked private-cache entries) will not incur an inordinate number of conflict and capacity misses. A directory unable to track all the contents of the private caches would lead to a steep increase of directory-induced invalidations, ruining the overall performance of the system. Hence, 100% coverage is a hard lower bound in the reduction of entry count if we want to build CMPs with reliable performance.

Our purpose in this chapter is to tackle the area scalability problems of directories containing an increasing number of entries from a novel perspective. We address an inefficiency that consists of the use of a fixed-size dedicated structure for the directory, typically made up of set-associative arrays. In this chapter, we will show that such structure is unnecessary when scaling up the number of cores, and it is actually harmful in a number of ways, as it introduces unaffordable area overhead and causes a high number of directory-induced invalidations. We discuss a novel way to store directory entries that we call In-Cache Coherence Information (ICCI). ICCI uses storage structures already present in the chip (the LLC) to dynamically store directory entries with fine granularity, allocating just the strictly necessary number of these. ICCI is based on the observation that as the number of cores rises, the number of directory entries required per-core does not increase, but instead it goes down on average thanks to data sharing.

The overhead of ICCI remains within acceptable limits for proper multi-core scalability regarding area, latency and energy. ICCI is orthogonal to the sharing code used (it is not our purpose to develop any new sharing codes). We will use full-map sharing codes for core counts up to 512, and a hierarchical code for larger core counts up to 256 K. The nature of coherence information makes ICCI very suitable to easily store it, rivaling in area with other scalable schemes (e.g., SCI, hierarchical protocols or duplicate-tag directories) without incurring their associated problems (e.g., complexity, slow invalidations) and providing some particular advantages instead, such as suffering a negligible amount of directory-induced invalidations.

ICCI provides dynamic directory coverage, which never rises over 100% but can be as low as 0% depending on runtime workload characteristics. We have studied the range of coverages resulting from different application characteristics and it can be concluded that, under reasonable circumstances, the total amount of storage used for storing coherence dynamically is within scalable limits.

The following background subsections describe the base architecture and the directory schemes needed to understand ICCI, and against which we have performed an evaluation based on detailed simulation. Usually, directory information co-located with the tags of the shared level of the cache hierarchy (Section 5.1.2) or sparse directories (Section 5.1.3), both based on full-map bit vectors, are the common schemes used for current CMPs of modest core counts. However, their scalability properties are not good, and new schemes are required to scale out coherence for large core counts.

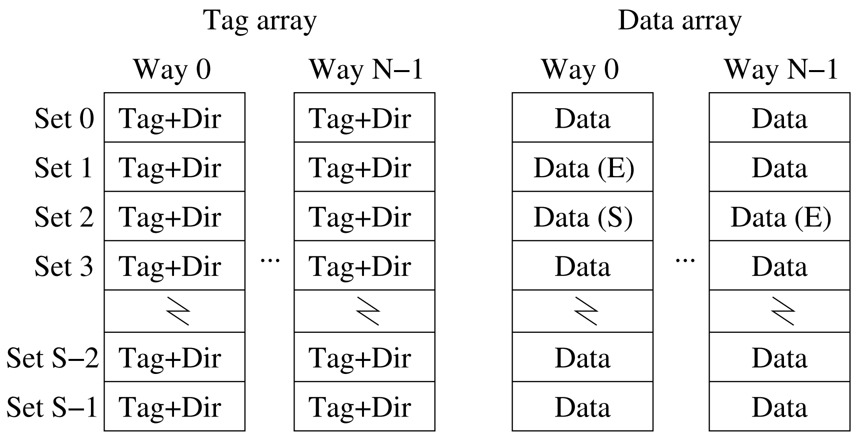


Figure 5.1: Tag-embedded sharing information.

5.1.1 CMP Architecture

The base architecture assumed in this chapter is the tiled-CMP described in Section 1.4. This tiled design uses NUCA caches [100], distributing the storage capacity across the chip, each tile containing a core and its associated bank of the shared NUCA last level cache (LLC). Through this chapter, we assume a cache hierarchy composed of a shared LLC and private L1 caches, without loss of generality. The cache coherence protocols discussed in this chapter maintain coherence among the private L1 caches, and use exact sharing information unless otherwise noted. All sharing information is stored within the chip for fast access, to keep track of lines in the private caches, instead of storing it in the slower off-chip DRAM. A scalable network (typically a mesh) connects all the cores. Next, we describe some cache coherence schemes necessary for understanding ICCI.

5.1.2 Inclusive Cache with Tag-Embedded Sharing Information

Inclusive cache hierarchies, in which each level of the hierarchy contains all blocks stored in the levels closer to the cores, provide a natural directory when sharing codes are co-located with the LLC tags [9, 34]. However, this option gets less interesting as the number of cores grows. Embedding the sharing code in the tags makes the entry size increase linearly with the number of cores. With 512 cores and a full-map bit-vector, the size of the sharing bit-vector

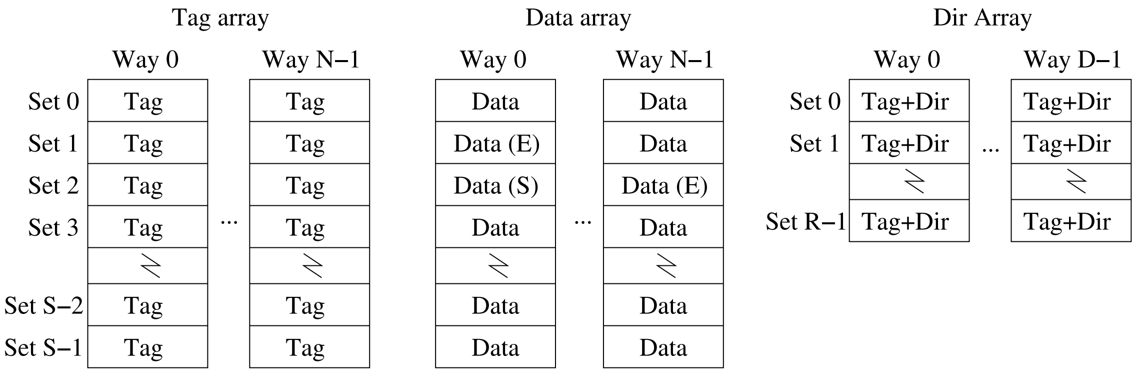


Figure 5.2: Sparse directory.

would match the size of its associated 64-byte cache line, and the total LLC area would almost double. This is an unaffordable overhead, and for this reason tag-embedded sharing information remains used only for low core counts. Intel's recent microarchitectures use this scheme to keep coherence among four private L2 caches, being the L3 cache shared and inclusive, with its tags containing core valid bits (CVB) to indicate which cores contain copies of the block in their private caches [130].

This scheme is depicted in Figure 5.1, for an LLC with S sets and N ways. Notice that all the entries of the tag array have room for storing sharing information (one bit per core in each entry), which is obviously wasteful, as only those blocks with copies in the private caches (represented with S and E in the figure) will make use of the sharing code. Most directory entries will end up unused, as the LLC typically has many more entries than there are entries to track in the private caches. Also, blocks in exclusive state in L1 caches (indicated with E in the figure), may contain stale data, and in any case, any request regarding them needs to be forwarded to the L1 cache that holds exclusivity for the block.

In general, if the total number of entries in the LLC is t times larger than in all the private caches combined (e.g., t is equal to 8 in Intel's Ivy Bridge [130]), this tag-embedded directory can potentially track $t \times n$ times as many sharers as there are single entries in the private caches, where n is the number of cores. This is an exorbitant capacity compared to the real usage of these resources.

5.1.3 Sparse Directory

Sparse directories [69] are caches that store sharing codes (instead of memory blocks) to track all the contents of the private caches. Since the L1 entries to track are much fewer than the LLC entries, the utilization of tag-embedded directories is low (i.e., most directory entries track no sharers), and the sparse directory takes advantage of this fact to reduce the coherence information overhead by storing a smaller number of sharing codes separately from the LLC. The sparse directory is banked and distributed following the same pattern as the NUCA LLC. Figure 5.2 depicts a sparse directory. Compared with Figure 5.1, $R \times D$ (the number of entries in the sparse directory) is typically a smaller value than $S \times N$, which accounts for the overhead difference between tag-embedded and sparse directories.

Sparse directories also enable non-inclusive and exclusive caches naturally. Blocks stored in the L1 caches are tracked by the sparse directory, and they do not need to be stored in the LLC, leaving room for extra blocks compared to inclusive caches. If a block can be stored in the LLC while copies in the private caches exist (tracked by the sparse directory), the cache hierarchy is said to be non-inclusive. If the block can be on either the LLC or the sparse directory, but not both, the cache hierarchy is exclusive. Exclusive and non-inclusive caches provide a more efficient use of LLC resources, at the cost of introducing extra 3-hop L1 misses (those in which, in addition to the request and response messages, a third message is needed to reach the L1 cache supplying the block), and performing data transmission on clean L1 writebacks (because no copy of the block exists in the LLC).

Directory-induced invalidations [51], L1 cache line invalidations performed on directory evictions to maintain coherence, are another serious problem in sparse directories. Because directory-induced invalidations affect blocks actively used by the cores, they generate extra L1 cache misses with negative effects on performance. To reduce the number of such invalidations, overprovisioned sparse directories are used, and 200%-coverages are not uncommon [37] —meaning that the sparse directory has twice as many entries as all the L1 caches that it tracks combined.

Notice that a 100%-coverage n-way sparse-directory, with n equal to the L1 cache associativity times the number of cores, would remove directory-induced invalidations altogether (if correctly managed). However, its excessively high associativity makes it impractical, mainly due to the resulting high energy consumption. Lower-associativity, overprovisioned sparse directories are the realistic alternative used instead. Even though smaller than a tag-embedded directory, a sparse directory has room for tracking $\frac{c}{100} \times n$ times the total number of entries in the private caches, where c is the coverage (e.g., 200%) and n the number of cores. However, this is still a large amount of resources devoted to store directory information, most of which will be always unused.

Sparse directories are not practical for storing exact sharing codes with large core counts because of their poor scalability. Their per-core size is proportional to the number of cores, just like tag-embedded directories.

5.1.4 Scalable Coherence Directory

Hierarchical directories [125] have been proposed as a scalable alternative to flat directories. By distributing the sharing information in several levels, their per-core overhead is proportional to the k-th root of the number of cores (where k is the number of levels of the hierarchy, fixed at design time). However, the complex management of the distributed sharing information makes these protocols difficult and costly to validate and implement in real systems.

The Scalable Coherence Directory (SCD) [159] was recently proposed as a way around the complexity of hierarchical coherence protocols, while retaining their area-overhead advantages. To date, the Scalable Coherence Directory (SCD) [159] is arguably one of the most promising directory schemes for supporting coherence for high core counts (see Section 5.4 for other recent proposals). SCD is capable of storing exact sharing information in hierarchical entries within a single cache. This way, SCD has the same scalability properties as hierarchical protocols, while avoiding their complexity. The strong point of SCD is that a flat directory can be used instead of a hierarchical one, because all sharing information of a memory block can be found in the same directory cache. A flat directory protocol is much simpler, easier to implement, and even to formally prove correct, than a hierarchical directory.

The smart sharing information encoding of SCD enables a memory block shared by few cores to use just one entry. Additional entries are allocated as the number of sharers increases, creating a tree structure, with entries pointing to child-entries containing further sharers.

On the other hand, the storage of sharing codes in multiple entries makes SCD's lookup mechanism more complex. Multiple lookups are needed to retrieve all the SCD cache entries containing the sharing information of highly shared blocks. On such common events as block invalidations, this multiple-lookup process is carried out to retrieve the sharers to be invalidated. In general, up to $\sum_{a=1}^{k} \frac{n}{(\sqrt[k]{n})^a}$ lookups are necessary to read all the entries of a k-level hierarchy, assuming n cores. For instance, a 512-core 2-level hierarchy requires 23 sequential lookups in the same SCD cache, in the critical path of the invalidation process.

Additional hierarchy levels reduce SCD's entry size but increase the number of lookups. SCD's entries for a 9-level 512-core hierarchy require just a 2-bit vector, but 511 of these entries make up the hierarchy. Reading so many entries sequentially for an invalidation is not affordable.

As long as the lookup latency of SCD overlaps with other latencies, it should not affect performance too much. Prior work tested SCD with a unicast network, and the results showed that the look-up latency for a 1024-core 2-level hierarchy was overshadowed by the latency of sending messages sequentially [159]. However, unicast is very inefficient when sending messages to large core counts. Mechanisms such as multicast or cruise-missile invalidates [12] are more appropriate then for both the sending of requests and the recollection of responses, as suggested by SCD's authors, to improve the speed and energy efficiency of the system. When using these mechanisms, the effect of the look-up latency of SCD on execution time would become more important.

As for directory-induced invalidations, SCD relies on the high-associative properties of ZCaches [158] to reduce their number while using very small coverages (e.g., 110%). ZCaches provide high associativity by considering many replacement candidates on evictions. This allows SCD to outperform traditional hierarchical protocols in which directory-induced invalidations are more frequent. Further details on SCD can be found in Sanchez et al. [159].

The scheme depicted in Figure 5.2 is applicable to SCD. In essence, SCD uses a directory cache with optimized composable sharing codes.

5.2 ICCI: In-Cache Coherence Information

ICCI is a new cache organization that provides natural support for storing cache coherence information. This support is derived from a novel usage of LLC entries to store either a cache line or sharing information about the copies of a memory block stored in the L1 caches.

The LLC is dynamically filled with cache lines and sharing codes, taking up just the strictly necessary number of entries for storing cache coherence information. A flat cache coherence protocol is used to maintain coherence. No specific sharing code is enforced by ICCI. We use a full-map bit vector along most of this chapter to make use of the large storage capabilities of ICCI, but other sharing codes can be used. More efficient compressed codes for larger core counts can be enabled by the large size of cache lines (or several LLC entries may be combined to create even larger composable codes, similarly to SCD).

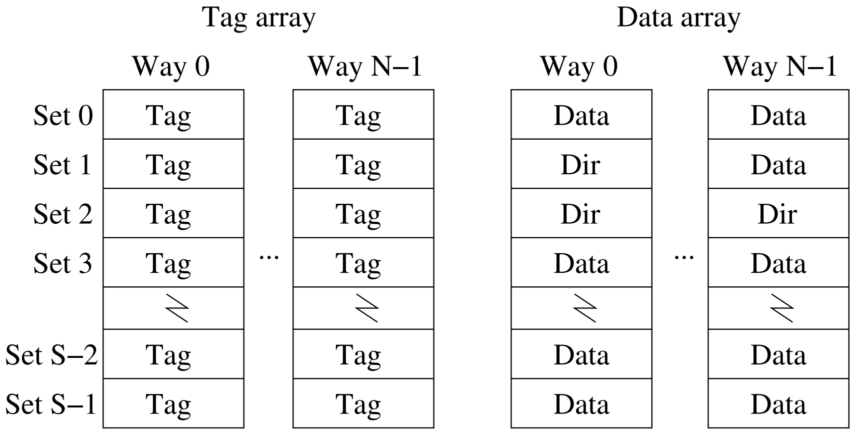


Figure 5.3: ICCI cache coherence.

Figure 5.3 depicts ICCI's way to organize the LLC and sharing information. To understand ICCI properly, it is illustrative to compare it with tag-embedded directories (Figure 5.1). We can think of it this way: ICCI moves the tag-embedded sharing code from the tag into the data field of the cache entry. The tag no longer needs to grow in size, while the data field has plenty of space to store large sharing codes (e.g., 512 bits with 64-byte cache lines). Because a cache entry cannot store a memory block when the data field is occupied by the sharing code, the system must be adapted to work with this kind of cache organization. Minor modifications to the LLC management are enough to enable ICCI's operation. The following subsection (Section 5.2.1) explains ICCI's operation in detail.

Comparing Figure 5.3 and Figure 5.2, we can see that ICCI provides a huge directory compared to the sparse directory ($N \times S$ is larger than $R \times D$). However overprovisioned, a reasonably sized sparse directory will always be smaller than the LLC, which means that directory-induced invalidations will be much less frequent in ICCI.

Comparing ICCI with Figure 5.1, we observe that ICCI stores sharing codes in entries that, in the tag-embedded directory, contain potentially stale data (marked with E). This data can be stale because the core caching it has write permissions and may have modified the data in its private cache, and thus a request for that data received by the LLC would need to be forwarded to the L1 cache that has the updated data both in the case of the tag-embedded directory and ICCI.

However, ICCI also stores sharing codes in entries that contained shared data in the case of the tag-embedded directory. Read requests for that data could be answered directly by the LLC in that case, while now in ICCI they need to be forwarded to the owner L1 cache.

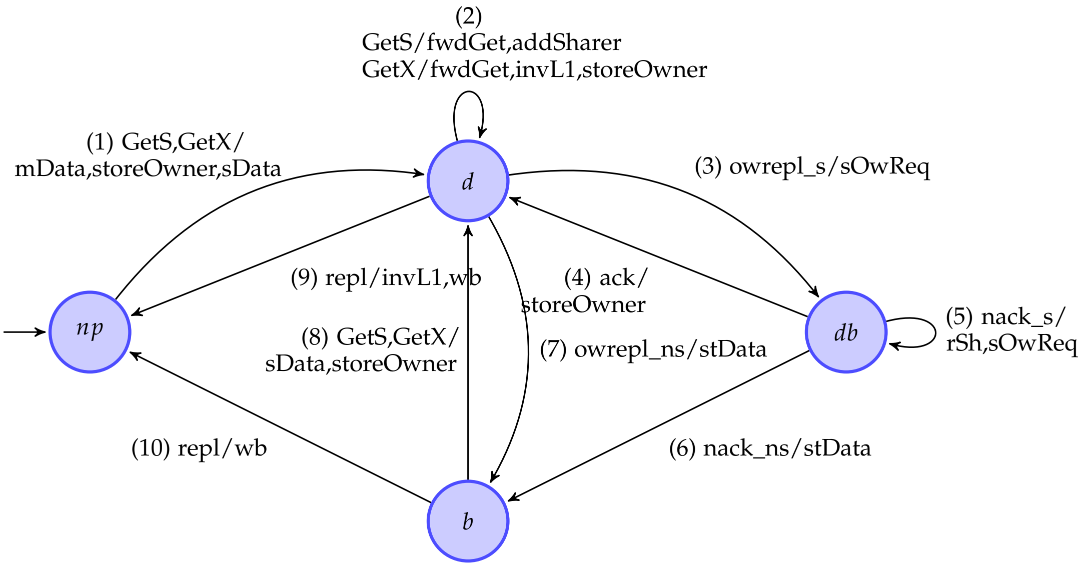
In addition, contrary to tag-embedded information and sparse directories, ICCI introduces no dedicated structures and no fixed directory overhead. Traditional directory schemes are designed to cover worst-case scenarios in both terms of number of entries and entry size, which makes them scale poorly. The fundamental idea behind ICCI is that the storage of directory information can be performed more efficiently (and more simply) if, rather than using a dedicated structure, the system takes up just the minimum amount of entries required from the LLC. As the number of cores increases, the number of directory entries really needed per core typically decreases (Section 5.3.4.1), making ICCI's approach scalable.

To conclude with the description of ICCI, we must note that the characteristics of ICCI can be seen from two complementary points of view, depending on how resources are assigned:

- Assuming a fixed LLC capacity. Contrary to tag-embedded or sparse directories, which in this case introduce large overheads for storing the sharing codes, ICCI introduces no extra overhead on the storage capacity of the chip. However, the usage of cache entries to store information in ICCI is reflected by changes in the characteristics of the cache hierarchy that are discussed in Section 5.2.4.

- Assuming a fixed amount of resources that are shared between cache blocks and coherence information. In traditional schemes, these resources are split statically between the LLC and the structures storing coherence information at design time, resulting in a smaller LLC (and increasingly smaller as the number of cores rises). This is not the case in ICCI, where the LLC will be assigned all the resources and they will be used dynamically to store either data or coherence information. ICCI will adapt at runtime to the characteristics of the applications running, changing the percentage of the resources used for sharing information depending on the sharing patterns of the workload at each moment.

In both cases, the most appropriate scheme will be determined by the most efficient global usage of resources. We use the first point of view for the detailed evaluation of ICCI against other proposals in Section 5.3. In Section 5.3.4, we use the second point of view to analytically show ICCI's favorable scalability



states

np: block not present
d: directory info present, block in L1 cache(s)
b: block in LLC only
db: seeking new owner

events

GetS: a core's load request
GetX: a core's write request
owrepl_ns: replacement from L1 (no additional sharers exist)
owrepl_s: replacement from L1 (additional sharers exist)
repl: LLC block replacement
nack_ns: L1 cannot accept ownership (no more sharers)
nack_s: L1 cannot accept ownership (sharers exist)
ack: L1 accepted ownership

actions

mData: get data from main memory
storeOwner: store owner in the directory information
sData: send data to requesting core
fwdGet: forward request to owner
addSharer: store new sharer in the directory information
invL1: invalidate L1 copies
sOwReq: send request for a sharer to accept ownership
rSh: remove stale sharer
stData: store data in the LLC
wb: send data to main memory

Figure 5.4: ICCI's state diagram for the LLC.

properties compared to other coherence schemes, depending on the particular characteristics of the running applications.

5.2.1 ICCI LLC Management

Figure 5.4 shows ICCI's operation state diagram for the LLC. An ordinary MOESI cache coherence protocol is assumed (others are possible). The states shown in the diagram correspond to the possible configurations of a block in the LLC: not present (*np* state), directory information stored in the LLC (*d* state), and block stored in the LLC (*b* state). These states are codified in the tag array of the LLC. Next, we give an explanation of these states.

When a block is fetched from main memory, an LLC entry is allocated for the sharing code (*d* state), and the block is only stored in the requesting core's L1 cache, which becomes the block owner (1). Other cores can get a shared copy of the block by sending a request to the LLC, which forwards the request to the owner L1 cache (2), which answers with the shared copy.

Only the owner core writes back the block data to the LLC upon eviction, while other cores' shared copies can be optionally replaced silently. When the owner replaces the block, the LLC asks another sharer (if any exists) to accept the ownership (3–4). Other sharers are known to the LLC thanks to the sharing code stored in the LLC. However, if a sharer has silently replaced its shared copy of the block, it will reject the ownership. In such a case, the LLC removes the former sharer from the sharing code and probes another sharer (5). This process is out of the critical path of L1 cache misses and is only needed if silent replacements are allowed. If there are no sharers left, the sharing code stored in the LLC entry is not necessary anymore, and the evicted memory block reuses that LLC entry (6/7), transitioning from *d* to *b* state. Reusing the LLC entry also prevents LLC evictions upon L1 cache replacements.

When a core requests a block stored in the LLC, the block is sent to the requesting core (in exclusive state), and the LLC entry that contained the block is reused to store the newly generated sharing code (8), transitioning from *b* to *d* state. Again, the entry reuse mechanism prevents any LLC evictions. This reuse is important, since directory entry evictions are the cause of directory-induced invalidations.

In ICCI, only main memory accesses (due to LLC misses, which are hopefully infrequent) cause LLC evictions of either a directory entry (9) or a block (10) in order to allocate a directory entry for the newly fetched block.

ICCI's LLC uses a pseudo-LRU replacement policy. The data blocks stored in the LLC never get their pseudo-LRU information updated in ICCI, because accesses to them cause their substitution by directory entries (8). Only entries containing directory information have their pseudo-LRU information updated by new accesses by cores. Hence, data blocks are commonly evicted from the LLC before directory entries naturally. In addition, sharing codes are not evicted as long as there are candidate blocks for eviction. This actually makes ICCI work implicitly as the mechanisms proposed by Jaleel et al. [91] to bridge the performance gap between inclusive and non-inclusive caches, which in practice are meant to reduce directory-induced invalidations.

5.2.2 Storage Efficiency

ICCI avoids the inefficient use of storage space caused by other directory schemes. Three are the sources of this inefficiency: tag duplication, fixed directory size and overprovisioning.

The first one appears when a block uses entries in both the LLC and the directory cache, using one tag in each structure. This waste is saved by ICCI, since it only uses one LLC tag for either the block or its sharing code.

The second one comes from the large fixed capacity of the directory cache, accounting for when all L1 blocks are private to the cores, each block being tracked by one directory entry. When a block is shared, one directory entry tracks all the sharers, resulting in other directory entries being unused. For instance, instructions can be widely shared, especially when running parallel applications. As the core count grows, this inefficiency increases. In general, the number of directory entries used in an n-core CMP dynamically ranges between the aggregate number of L1 lines —when all L1 lines are private— and just $\frac{1}{n}$ of that value —when all L1 lines are shared by all cores—. Hence, a 512-core CMP may use as few directory entries as $\frac{1}{512}$ of the maximum number at times.

The third inefficiency aggravates the second. It is common to oversize the directory cache to reduce directory-induced invalidations. Reported coverage ratios of 200% [37] imply that at least half of the directory cache is empty at all times, even when all L1 blocks are private.

ICCI, on the contrary, gives all these otherwise unused resources to the LLC. ICCI's unified storage allows the number of directory entries stored in the LLC to grow or shrink dynamically as required by applications. When fewer directory entries are used, more memory blocks are accommodated in the LLC. Section 5.3.4 explores this topic in depth.

In conclusion, ICCI incurs none of the typical inefficiencies of directory schemes, and given an equivalent amount of area resources, it uses them more efficiently.

5.2.3 Contextualizing ICCI's Directory Scheme

Table 5.1 compares ICCI with other shared-memory organizations based on cache coherence directory protocols, showing that ICCI has the best features among them. For the moment, consider ICCI with a full-map bit vector (ICCI with SCD's sharing code will be evaluated in Section 5.3.4). Especially important is ICCI's ability to store directory information with no additional area overhead,

Table 5.1: Comparative of directory schemes.

	Directory Features			
Scheme	**Type**	**Per-tile overhead**	**Lookup latency**	**Directory-induced invalidations**
Tag-embedded	Flat	O(n)	O(1)	Low
Sparse directory	Flat	O(n)	O(1)	Medium
Hierarchical	Hierarchical	$O(\sqrt[k]{n})$	$O(\log n)$	High
SCD	Flat	$O(\sqrt[k]{n})$	$O(\frac{n}{\sqrt[k]{n}})$	Low
SCI	List-based	$O(\log n)$	O(n)	Medium
Pointer Tree	Tree-based	$O(\log n)$	$O(\log n)$	High
ICCI-full-map	Flat	none	O(1)	Low
ICCI-SCD	Flat	none	$O(\frac{n}{\sqrt[k]{n}})$	Low

	Influence in the complexity of	
Scheme	**Coherence protocol**	**Cache design**
Tag-embedded	Low	Low
Sparse directory	Low	Low
Hierarchical	High	Low
SCD	Low	High (hierarchy on ZCache)
SCI	Medium	Low
Pointer Tree	High	Low
ICCI-full-map	Low	Low
ICCI-SCD	Low	Medium (hierarchy on standard cache)

unlike the rest of schemes. Also, ICCI's negligible number of directory-induced invalidations (Section 5.2.3.1) contrasts with most schemes that at some degree suffer performance degradation due to these invalidations. Other good features of ICCI are the use of a simple flat protocol and ordinary caches, as well as its constant lookup latency, especially if compared with the closest alternative, SCD.

Finally, note the differences between ICCI and an apparently similar published proposal: AMD Magny-Cours' cache coherence [37]. When operating in non-coherent mode, the Magny-Cours uses all the ways of the cache to store blocks. When operating in coherent mode, the Magny-Cours allocates some ways of all cache sets to work as an ordinary directory cache. This is different from ICCI because while the Magny-Cours creates a separate directory cache, ICCI selectively uses any LLC entry to store sharing information. Besides, ICCI also allows operation in non-coherent mode, allocating all the ways of the cache to store blocks.

5.2.3.1 Directory-Induced Invalidations

ICCI incurs a negligible amount of directory-induced invalidations. In ICCI, the eviction of directory entries from the LLC, causing L1 cache invalidations, is a rare phenomenon due to the much larger size of the LLC compared to the tracked L1 caches and the fact that data blocks are evicted from the LLC before directory entries. For instance, ICCI working on an LLC with 8 times as many entries as the aggregate L1 caches works logically as an 800%-coverage directory cache. Applying the analytic model proposed by Sanchez et al. [159] to ICCI, the maximum probability of evicting a cache entry with sharing information upon an LLC insertion is $(\frac{size_{L1}}{size_{LLC}})^{assoc_{LLC}}$. Assuming the previous ratio between LLC and private caches and 8-way associativity in the LLC, the eviction probability is 6×10^{-8} for ICCI. As a comparison, SCD using overprovisioned 110%-coverage 64-replacement-candidate ZCaches has a much larger 10^{-3} eviction probability, which is considered negligible by SCD's authors. Moreover, ICCI's eviction probability is applicable only upon memory accesses, because it is then that LLC insertions take place (see how entry reuse works in Section 5.2.1), while for SCD evictions take place upon more frequent LLC accesses (SCD entries are allocated for the directory information of the LLC blocks accessed by the cores, causing evictions). Even if, as noted by SCD's authors, empirical results for set-associative caches do not exactly match the analytical model due to non-uniform distribution of blocks (compared to ZCaches and their elaborated hashing strategies), several orders of magnitude of margin exists between ICCI's and SCD eviction probabilities, and our experimental results confirm that ICCI shows several order of magnitude fewer directory-induced invalidations. Note that one of the main contributions of SCD is its ability to bound the eviction probability by means of controlled overprovisioning thanks to ZCache's high associativity. We have shown that ICCI can do as good a job with no need for the (overprovisioned) complex ZCache-based SCD cache. This also enables the use of SCD's sharing code with ICCI and ordinary set-associative caches (Section 5.3.4.2).

5.2.4 Contextualizing ICCI's Cache Hierarchy

To understand the performance numbers of the evaluation section, the differences between cache hierarchies must be taken into account. Table 5.2 compares typical cache hierarchies and ICCI. Three features are shown: 3-hop misses, meaning when an access to the LLC must be forwarded to an L1 cache containing the

Table 5.2: ICCI and other cache hierarchies.

	3-hop misses	Clean data writebacks	Number of memory accesses
Inclusive (TAG)	exclusive L1 blocks	no	high
Non-Inclusive (SPARSE, SCD)	owned L1 blocks	yes	medium
Exclusive	all L1 blocks	yes	low
ICCI	all L1 blocks	yes	high

block; clean data writebacks, meaning if a clean cache line evicted from an owner L1 cache must be sent back to the LLC to store the data there; and number of memory accesses. Each cache hierarchy possesses a different combination of these features.

ICCI incurs as many 3-hop misses as exclusive caches do, because none of them store in the LLC lines which are already present in the L1 caches. In ICCI, clean blocks evicted from L1 caches are written back to the LLC (transitions 6 and 7 in Figure 5.4), like in non-inclusive and exclusive caches, because the block is not stored in the LLC while owned by an L1 cache. Finally, ICCI works as an inclusive cache regarding the number of memory accesses, because blocks in L1 caches require an LLC entry for the sharing code, while non-inclusive and exclusive caches provide a higher effective total cache capacity.

5.3 Evaluation

We used a simulator based on Pin [118] and GEMS [126] to perform the tests shown in this section. The chip components of GEMS were attached to a Pin tool to enable fast simulation of large numbers of cores. The methodology explained by Monchiero et al. [131] was used to obtain performance numbers.

5.3.1 Parameter Settings

We simulated a 512-core CMP (shown in Figure 5.5) running at 2 GHz with a shared 8-way associative L2 cache based on a NUCA design (one 10-cycle access latency 256 KB L2 bank per core, 128 MB total) on a mesh network (every two tiles share a router), and 4-way associative 16 KB data and instruction L1 caches (1-cycle access latency). The cache block size is 64 bytes. We call the capacity ratio between the shared cache and the aggregate private caches as the S/P ratio.

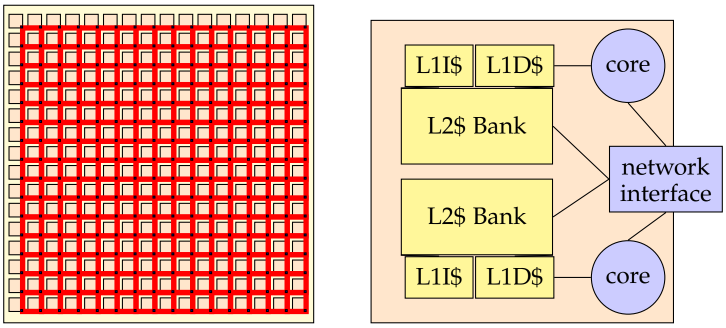


Figure 5.5: 512-core CMP. Two tiles (each containing a core, private L1 caches and a shared L2 cache bank) share each router of the 16×16 mesh network.

The S/P ratio of our simulations was 8×. For main memory, we assumed DDR4 technology [111, 165].

For reference, the cache sizes were set to those of Intel's SCC [81], which was designed to scale out to hundreds of cores. Intel's SCC measures 567 mm^2 at 45 nm with a 125 W TDP. Our assumed 512-core could be realized in 585 mm^2 at 14 nm with similar power consumption.

For energy calculations, we used McPAT [112] assuming a 22 nm process and scaled the resulting figures down to 14 nm. Simulations were configured with the values generated by McPAT. Table 5.3 summarizes the characteristics of the simulated machine.

We evaluated four different schemes. Two of them are common area-consuming (non-scalable) directory coherence schemes to use as baselines: a tag-embedded directory (TAG) and a 200%-coverage sparse directory (SPARSE). The other two are the scalable directory coherence proposals we intend to compare: a 110%-coverage SCD and ICCI. Table 5.4 shows the simulated directory schemes and the overhead of their associated extra resources. Notice that ICCI is the only alternative that requires no additional hardware resources, while the tag-embedded directory requires the most resources.

The 110%-coverage SCD uses 52-replacement-candidate ZCaches. We used ordinary 200% and 110% coverages for the sparse directory and SCD to reduce directory-induced invalidations to a negligible number [37, 159]. The sparse directory needs a higher coverage to achieve a similar number of directory-induced invalidations to SCD, as SCD takes advantage of the higher associativity of ZCaches. The tag-embedded directory uses an inclusive cache hierarchy, while the sparse directory cache and SCD allow for a non-inclusive hierarchy, and ICCI's cache hierarchy has its own characteristics (see Section 5.2.4 for

Table 5.3: Simulated machine

Processors	512 x86 cores @ 2 GHz, 2-ways, in-order
L1 Cache	Split I&D. Size: 16 KB, 4-ways, 64 bytes/block Access latency: 1 cycle MOESI coherence protocol
L2 Cache	Size: 256/128/64 KB per bank (NUCA) 16-ways, 64 bytes/block Access latency: 10 cycles Directory cache lookup: 2 cycles, SCD cache lookup: 1 cycle
RAM	16 GB DDR4 DRAM 16 3D-stacked memory controllers
Interconnection - Mesh	2 GHz, 2D mesh: 16×16. Express links every 4 routers 16 byte links Latency: 1 cycle/link, 3 cycles/express-link 4-cycle pipelined routers Flit Size: 16 bytes Control/Data packet size: 8/72 bytes (1/5 flits)

Table 5.4: Directory size requirements for the schemes tested. Size is given as a percentage of the aggregate capacity of the tracked caches, assuming a 512-core CMP and 64-byte lines.

S/P ratio	Embedded-tags	200%-coverage sparse directory	110%-coverage SCD	ICCI
8×	729%	200%	15%	0%
4×	364%	200%	15%	0%
2×	182%	200%	15%	0%

details). This involves associated effects that we discuss later. Both TAG and ICCI implement a suitable LLC replacement algorithm to minimize the performance loss of inclusive caches with respect to non-inclusive ones [91] (see Section 5.2.1).

Note that due to the difficulty in simulating and implementing arbitrary cache sizes (i.e., sizes not power of two), we did not fix the overall amount of area resources and derive the sizes of the LLC and directory storage. Instead, we fixed the LLC size and added the extra resources needed by each directory scheme.

ICCI removes the need to access a directory cache in addition to the L2 tags. Note that the directory cache causes extra energy consumption if accessed in

Table 5.5: SPLASH-2 program sizes.

Benchmark	**Original problem size (maximum 64 cores)**	**Scaled problem size (512 cores)**
barnes	16K particles	256K particles
ocean_cp	258×258 grid	2048×2048 grid
ocean_ncp	258×258 grid	2048×2048 grid
volrend	ROTATE_STEPS=4	ROTATE_STEPS=100
water_ns	512 molecules	8K molecules
water_s	512 molecules	32K molecules
cholesky	tk29.O	tk29.O
fft	64K points	1M points
lu_cb	512×512 matrix	2048×2048 matrix
lu_ncb	512×512 matrix	2048×2048 matrix
radix	256K integers	64M integers

parallel to the LLC or extra latency if accessed sequentially after the LLC. We have considered parallel accesses to maximize performance.

We ran benchmarks from the SPLASH-2 suite appropriately scaled up for 512 cores. Table 5.5 compares the original SPLASH-2 input sizes recommended for up to 64 cores and the scaled-up input sizes used in our experiments for 512 cores.

We considered the use of unicast or multicast networks in the simulated 512-core chip. Our preliminary results showed that unicast communication causes performance to drop in all benchmarks compared to using efficient one-to-many and many-to-one communication, as noted by Ma et al. [120]. We observed that seven out of eleven evaluated benchmarks increased their execution time by 50% at least when using unicast communication. The least affected benchmark was fft, which still showed a 5% increase in execution time. The main cause is the slow invalidation of highly shared blocks, which becomes a bottleneck and increases the pressure on the network creating hot spots. While multicast can gracefully deal with invalidations to many cores, which are especially important for efficient thread synchronization (e.g., barriers and locks), unicast requires the origin of invalidations to send up to 511 unicast messages and process up to 511 response messages, becoming a fatal bottleneck for performance, as evidenced by our results. For its superior performance, we chose a network with efficient multicast request sending and response collection to evaluate the four directory schemes [120].

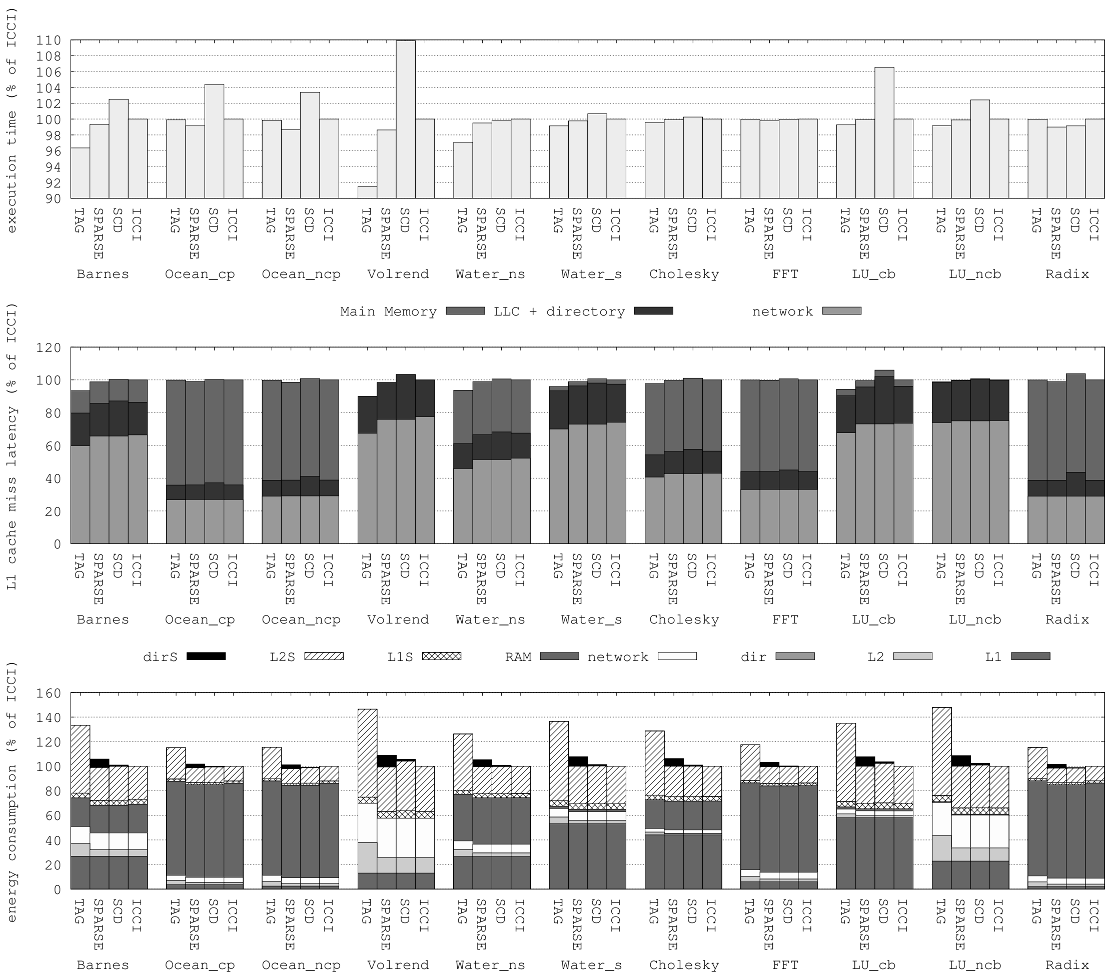


Figure 5.6: Results for 8× S/P ratio. From top to bottom: execution time, average memory access time and energy consumption.

5.3.2 Results for 8× S/P Ratio

5.3.2.1 Execution Time

The top graph of Figure 5.6 shows the execution time of the SPLASH-2 benchmarks. The results are normalized to ICCI. The central graph of the figure gives insight into how time is spent on L1 cache misses, showing the main differences between the four evaluated directory organizations.

In general, ICCI works similarly to the slower cache hierarchy in each benchmark: inclusive (TAG) or non-inclusive (SPARSE). ICCI suffers as many extra

Table 5.6: LLC miss rate.

	8× S/P ratio		
Benchmark	**Non-Inclusive**	**ICCI**	**Increase**
barnes	3.3%	3.4%	3.8%
ocean_cp	35.3%	35.8%	1.5%
ocean_ncp	30.9%	31.6%	2.4%
volrend	0%	0%	0%
water_ns	10.5%	10.5%	0.5%
water_s	0.6%	0.6%	0%
cholesky	16%	16%	0.3%
fft	25.3%	25.4%	0.5%
lu_cb	0.9%	0.9%	0%
lu_ncb	0.1%	0.1%	0%
radix	31.1%	31.7%	1,8%

memory accesses as the inclusive cache used by TAG, increasing execution time in ocean, fft and radix. ICCI suffers even more extra 3-hop accesses than the non-inclusive hierarchy used by SPARSE, increasing the execution time of barnes, volrend and water. Volrend is the best benchmark for TAG, with 7% and 8% faster execution than SPARSE and ICCI at 8× S/P ratio, due to the difference in number of 3-hop misses. At 8× S/P ratio, ICCI performs less than 2% worse than the fastest coherence scheme in 8 out of 11 benchmarks.

As for SCD, it performs similar to SPARSE since both use a non-inclusive hierarchy. This is an advantage over ICCI when memory accesses make up most of the execution time, like in radix, as SPARSE and SCD reduce the LLC miss rate compared to TAG and ICCI.

However, SCD's weak point is the multiple sequential directory lookups required to reconstruct the sharing vector that take place in the critical path of cache misses. The effect of these accesses is especially harmful when they take place in critical events such as barriers or contended locks, affecting the performance of many cores, increasing the inefficiency of thread synchronization. This makes SCD results deviate from those of SPARSE in several benchmarks. This is the case in ocean, volrend and lu, in which SCD shows degraded performance, with up to a 10% slowdown in volrend compared to ICCI.

In broad terms, both ICCI and SCD perform reasonably close to the non-scalable tag-embedded directory and sparse directory, with the performance differences just described. Their worse performance in some cases can be justified because they use far fewer resources than the non-scalable organizations (see

Table 5.4). SCD's degraded performance in some benchmarks (ICCI beats SCD in 8 out of 11 benchmarks) as well as ICCI's simplicity and smaller area are the main arguments in favor of ICCI.

These results suggest that ICCI's cache hierarchy features (see Section 5.2.4) do not affect execution time very negatively compared to other hierarchies, obtaining better results than other scalable alternatives like SCD. We went into detail to investigate why this is so, and summarize our findings next.

First, in ICCI as well as in inclusive caches (TAG), most of the LLC resources are used to store memory blocks not present in L1. In general, the total amount of L1 entries is much smaller than that of LLC entries, hence only a small percentage of the LLC is used for blocks present in L1 (or for sharing information in the case of ICCI). In addition, the least re-referenced blocks of the exclusive LLC are the ones not present in the inclusive LLC, and these cause few extra memory accesses. Hence, the LLC miss ratio increase is small. Table 5.6 compares the LLC miss rate of ICCI and the non-inclusive cache used with SCD. In general, some benchmarks have a working set that fits in both the non-inclusive LLC and ICCI, both yielding the same miss rate. Some benchmarks have large working sets and the miss rate of ICCI is higher.

These results agree with the general observation that LLC miss rate is approximately inversely proportional to the square root of the effective size of the LLC [77]. In addition, prior work showed that the performance gap between inclusive and non-inclusive caches is caused by inappropriate management of LLC replacement information in inclusive caches, rather than by the difference in effective LLC capacity, and appropriate replacement policies can bridge that gap almost completely [91]. ICCI was designed in such a way that an appropriate update mechanism is part of its operation (as was shown in Section 5.2.1), ensuring a performance close to non-inclusive caches.

Second, most 3-hop misses are caused by accesses to blocks currently owned by L1 caches, and they also take place in non-inclusive caches. As a result, the difference in the amount of 3-hop misses between ICCI and non-inclusive caches is small, as the results in Table 5.7 show. However, when 3-hop misses are abundant, the performance of a non-inclusive cache and ICCI can degrade compared to an inclusive one. This becomes evident in barnes and volrend when comparing the execution time and L1 miss latency shown in Figure 5.6.

Third, ICCI reduces the amount of data writebacks compared to a non-inclusive cache, as can be seen in Table 5.8. The explanation to this lies in ICCI's replacement mechanism, which transfers the ownership to another sharer upon an owner replacement. This means that clean data is not replaced as long as there

Table 5.7: 3-hop misses as a percentage of all L1 cache misses.

Benchmark	Non-Inclusive	ICCI	Increase
Barnes	59.1%	66.2%	11.9%
Ocean_cp	2.4%	2.5%	4.8%
Ocean_ncp	2.8%	3.5%	26.3%
Volrend	75.5%	89.9%	19.2%
Water_ns	71.5%	83.6%	16.8%
Water_s	26.4%	35.2%	33.3%
Cholesky	31.4%	34.4%	9.5%
FFT	0.3%	0.3%	0.4%
LU_cb	47.6%	51.2%	7.7%
LU_ncb	8.5%	9.8%	15.3%
Radix	0.5%	0.6%	13%

Table 5.8: Data writebacks (as a percentage of L1 writebacks).

Benchmark	Non-Inclusive	ICCI	Decrease
Barnes	40.8%	33.8%	17.3%
Ocean_cp	97.6%	97.4%	0.2%
Ocean_ncp	97.1%	96.4%	0.8%
Volrend	24.5%	9.9%	59.2%
Water_ns	28.4%	16.4%	42.4%
Water_s	74.4%	65.2%	12.3%
Cholesky	68.9%	65.7%	4.6%
FFT	99.6%	99.6%	0%
LU_cb	52.3%	48.7%	7%
LU_ncb	90.4%	89.1%	1.5%
Radix	99.5%	99.4%	0.1%

are sharers remaining in the chip. We found that ICCI generates traffic closer to the inclusive cache in this regard.

5.3.2.2 Energy Consumption

The bottom graph of Figure 5.6 breaks down the energy consumption of the memory system (including the interconnection network) normalized to ICCI. Static and dynamic energy consumption is taken into account for the caches, network-on-chip and RAM. Cache energy is broken down into static and dynamic energy to analyze the detailed effects of the directory area overhead, while RAM and network energy are not broken down for clarity.

These results show the inefficiency of embedded-tag directories and sparse directory caches. TAG approximately doubles the static energy of the LLC compared to ICCI, due to the directory information that is as large as the associated LLC block. The L2 cache dynamic energy of TAG also increases due to the larger tags, raising its total energy consumption even more. This results in increases in the overall energy of the memory system of up to 48% with respect to ICCI. SPARSE reduces the static energy compared to TAG thanks to the smaller area overhead of the directory cache. However, SPARSE still increases energy consumption in general when compared to ICCI, and does so by up to 15% in volrend. SCD reduces the static energy further compared to SPARSE, but its area overhead still makes it more energy consuming than ICCI.

In conclusion, ICCI is the least energy consuming alternative at an 8× S/P ratio, outperforming SCD in most benchmarks, and doing so clearly on those with moderate RAM usage. ICCI's lower execution time when SCD suffers from directory serialization causes the largest energy differences. This is especially noticeable in volrend, where ICCI reduces energy by 8% compared to SCD. ICCI consumes as much RAM energy as TAG, which is more than SPARSE and SCD, but the absence of area overhead clearly makes up for the increased RAM energy consumption. ICCI never consumes more energy than SCD at this S/P ratio.

5.3.3 Results for Lower S/P Ratios

To explore the effects of ICCI's increased pressure on the LLC for different S/P ratios, we simulated two smaller LLC sizes, maintaining the L1 cache size. In particular, we tested per-core L2 cache sizes of 128KB (S/P ratio of 4×) and 64KB (S/P ratio of 2×). For these ratios, the overhead of the coherence schemes tested can be found in Table 5.4. As a comparison, Intel's Sandy/Ivy Bridge microarchitectures have an inclusive shared L3 cache (8MB total) and private non-inclusive L2 and L1 caches (256 KB plus 32+32 KB per core, with 4 cores), resulting in an S/P ratio that varies from 8× to around 6.4×, depending on the degree of inclusivity at runtime between L2 and L1 caches. Note that a ratio of 2× should be rare and it is included just as a worst case scenario for ICCI.

Results for 4× and 2× S/P ratios are shown in Figures 5.7 and 5.8, respectively. They are in line with the results for the 8× S/P ratio previously discussed, and they also show the increased effect in LLC pressure of TAG and ICCI. The worst benchmark for ICCI (and TAG) is radix, which executes 6% more slowly than with SPARSE and SCD at a 2× S/P ratio, due to extra memory misses.

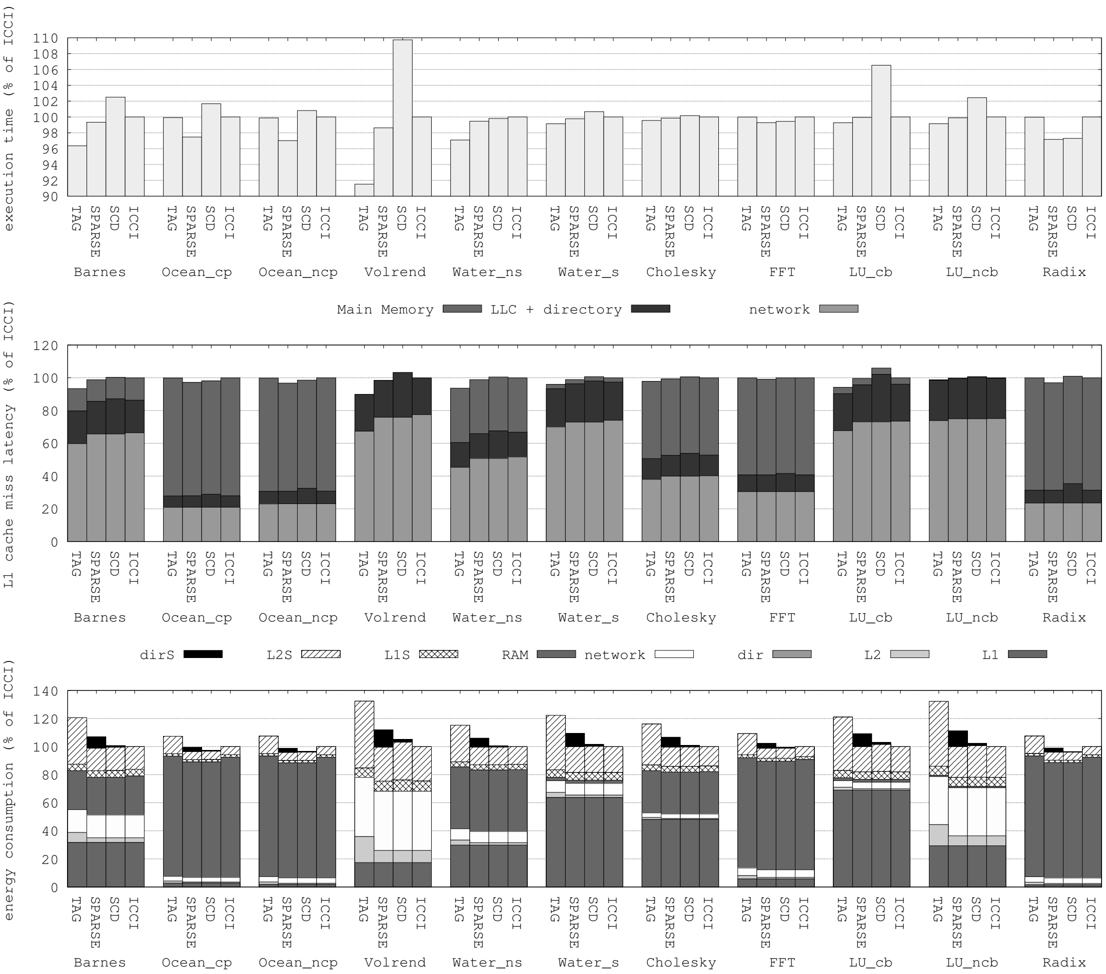


Figure 5.7: Results for 4× S/P ratio. From top to bottom: execution time, average memory access time and energy consumption.

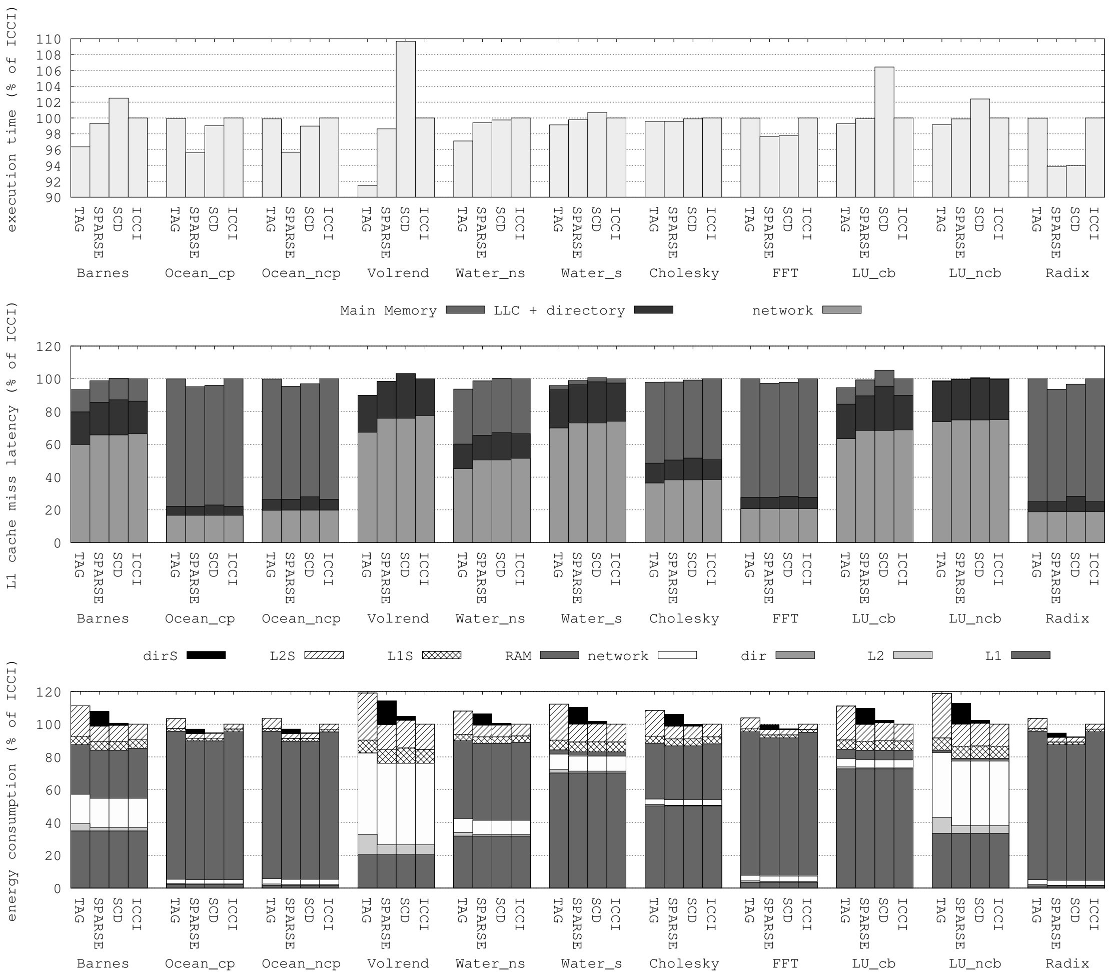


Figure 5.8: Results for 2× S/P ratio. From top to bottom: execution time, average memory access time and energy consumption.

Table 5.9: LLC miss rate.

	4× S/P ratio		
Benchmark	**Non-Inclusive**	**ICCI**	**Increase**
barnes	4.1%	4.2%	4%
ocean_cp	49.6%	51.7%	4.1%
ocean_ncp	43%	45%	4.8%
volrend	0%	0%	0%
water_ns	10.9%	11%	0.7%
water_s	0.6%	0.6%	0%
cholesky	18.4%	18.6%	1.1%
fft	28.7%	29.1%	1.5%
lu_cb	0.9%	0.9%	0%
lu_ncb	0.5%	0.5%	0%
radix	41.8%	43.7%	4.6%

Table 5.10: LLC miss rate.

	2× S/P ratio		
Benchmark	**Non-Inclusive**	**ICCI**	**Increase**
barnes	5.2%	5.5%	4.1%
ocean_cp	66%	70.3%	6.6%
ocean_ncp	52.4%	55.9%	6.6%
volrend	0%	0%	0%
water_ns	11%	11.2%	1.2%
water_s	0.6%	0.6%	0%
cholesky	19.6%	20.4%	3.8%
fft	50.6%	52.5%	3.9%
lu_cb	2.3%	2.4%	3.3%
lu_ncb	0.8%	0.8%	2%
radix	54.7%	59.9%	9.4%

Tables 5.6, 5.9 and 5.10 show that, as the S/P ratio decreases, the miss ratio difference increases between the non-inclusive caches (SPARSE and SCD) and ICCI, as expected. But even with a 2× S/P miss ratio, the largest miss ratio increase is 9.4% in radix. Those benchmarks with the largest increases are those in which the inclusive cache performs worse, with ICCI also suffering from higher LLC miss ratios.

In terms of energy, as the S/P ratio goes down, ICCI struggles to keep up in memory demanding benchmarks, with a clear 7% energy increase over SCD in radix at a ratio of 2× as the worst result for ICCI. Nevertheless, even with a 2× S/P ratio, ICCI reduces energy consumption in 6 out of 11 benchmarks compared to SCD. These results for such an unusually small S/P ratio and RAM demanding benchmarks show that ICCI makes a good job in containing energy consumption in very adverse conditions. Also, SPARSE power consumption gets closer to TAG as the S/P ratio goes down due to the shrinking difference in overhead between both schemes.

5.3.4 Exploratory Analysis of ICCI's Resource Usage

The comparison of directory schemes is complex when they result in cache hierarchies with different trade-offs in terms of LLC misses, 3-hop misses and write-back behavior, as is the case of TAG, SPARSE, SCD and ICCI. Detailed simulations allowed us to evaluate these schemes in the previous section, showing that ICCI has better overall performance features than the rest for the particular machine modeled in that section, taking into account execution time and energy consumption.

Additionally, the efficiency in the usage of resources by different directory schemes can be directly compared by measuring the storage resources taken up by each of them, in a way that the variations in the rest of the storage-dependent characteristics of the memory organization are eliminated (e.g., LLC miss rate). For instance, separate directories (such as SPARSE) take up a part of the resources, at design time, reducing the available space for the LLC. In contrast, ICCI takes up entries from the LLC on demand. The scheme using the least resources for directory information in practice will be the most beneficial from a point of view of storage, as in the end it will leave more resources available for other elements of the chip.

To carry out this analysis, first we have to distinguish between two types of directory coverage. We define the term *effective coverage* as the percentage ratio of the minimum number of directory entries required for tracking all the data in the private caches to the total number of tracked private-cache entries (e.g., a 100% effective coverage means that each private-cache entry requires one directory entry, and a 50% effective coverage means that, on average, every two private-cache entries contain the same block and are tracked by the same directory entry, hence requiring half as many directory entries as a 100% coverage). By definition, the effective coverage can never rise over 100%. On the other hand, we define the *physical coverage* of a directory as the ratio of the number of entries allocated for storing sharing information to the number of private-cache entries, whether the directory entries are currently in use or empty. The physical coverage of separate directories typically rises over 100%, like in 200%-coverage sparse directories.

The flexible allocation of entries in ICCI makes its physical coverage dynamically match the effective coverage at all times. For instance, if all blocks are widely shared by all the cores of a 512-core chip, ICCI's physical coverage becomes just $\frac{1}{512}$ of its physical coverage when all blocks are private. This also implies that the resources taken up by ICCI to store directory information vary dynamically in accordance. In other words, ICCI allows the directory to be resized at runtime to match the minimum necessary coverage required by the current workload, while the rest of resources are used to store data blocks. This is not possible in separate directories such as SPARSE or SCD, where the physical coverage is decided at design time, sizing the directory to fit every possible worst-case scenario. Fixed-size directories result in large amounts of resources assigned to a directory that will rarely be highly used at runtime, when effective coverages over 100% can never take place, and (much) smaller effective coverages often take place, especially as core count rises. In these rigid schemes, varying effective coverages simply translate into varying occupation rates of the fixed-size directory (i.e., the amount of directory entries that actually store sharing information). And despite whether the entries of the directory store sharing information or not at runtime, they cannot be used for other purposes (to our knowledge, no proposals do such thing yet), ending up wasted. Maybe the biggest contrast is that, while in ICCI the physical coverage never rises over 100%, in fixed-size directories physical coverage never goes under 100% (and a 100%-coverage directory is optimistically small and will in all probability yield bad performance due to conflicts that cause directory-induced invalidations).

On the other hand, it can be argued that ICCI wastes more space inside each directory entry compared to SPARSE. This is especially true for low core counts, when an LLC entry is obviously much bigger than a sharing vector (more on this and how to fix it in Section 5.3.4.2). First, this is not a particular problem of ICCI. In SPARSE, space is also wasted inside entries (e.g., if a large full-map bit-vector stores just one sharer, when a pointer would suffice). ICCI just accentuates this for low core counts, which does not imply that ICCI performs badly with small core counts. In fact, additional detailed simulations show that ICCI routinely outperforms a 200%-coverage sparse directory even for just 16 cores in both execution time and energy consumption (even though the sparse directory uses extra resources for the separated directory). This is so thanks to ICCI's better features such as lower number of directory-induced invalidations. Second, ICCI and SPARSE become more and more similar in entry size as the core count rises and the sharing code size approaches the LLC entry size.

Orthogonally to the detailed simulation perspective provided earlier, in this section we perform a wide exploratory survey of possible runtime characteristics to evaluate the real performance in terms of area of ICCI (i.e., the number of entries dynamically taken up for directory information), and we base our final assessments on typical effective coverages found in the literature. We carry out a theoretical analysis of the characteristics of ICCI, focusing on a system with a fixed amount of storage resources and evaluating the distribution of these resources between data and sharing information by a number of coherence schemes, with special emphasis on their scalability when scaling out CMPs to large core counts. The particular results will be very dependent on effective coverages and directory entry size.

This analysis only takes into account the area overhead of the evaluated directories. It does not measure other characteristics of the directory scheme such as energy consumption (e.g., huge in duplicated-tag directories for large core counts), latency (e.g., huge in SCI to go through the list of sharers upon invalidations), or other performance considerations (e.g., directory-induced invalidations, in whose prevention ICCI easily beats the rest). As ICCI forces no particular sharing code, we use a full-map bit-vector in the analysis for small core counts (up to 512 cores), and more elaborate sharing codes, in particular SCD's hierarchical code, for very large core counts (up to 256 K cores).

After this evaluation, we conclude that ICCI has good scalability properties in terms of area compared to other schemes. With feasible small coverages, ICCI's directory storage space is comparable to schemes such as SCI or duplicate-tag directories. In addition, in the worst case scenario for ICCI, when the effective coverage is 100%, it is still much more area-efficient than SPARSE for large core counts, using the same full-map bit-vector sharing code. Also, we will explore ways to turn this worst-case scenario into a best-case scenario (with potentially 0% area overhead) thanks to ICCI's dynamic coverage that can leverage complementary coherence mechanisms.

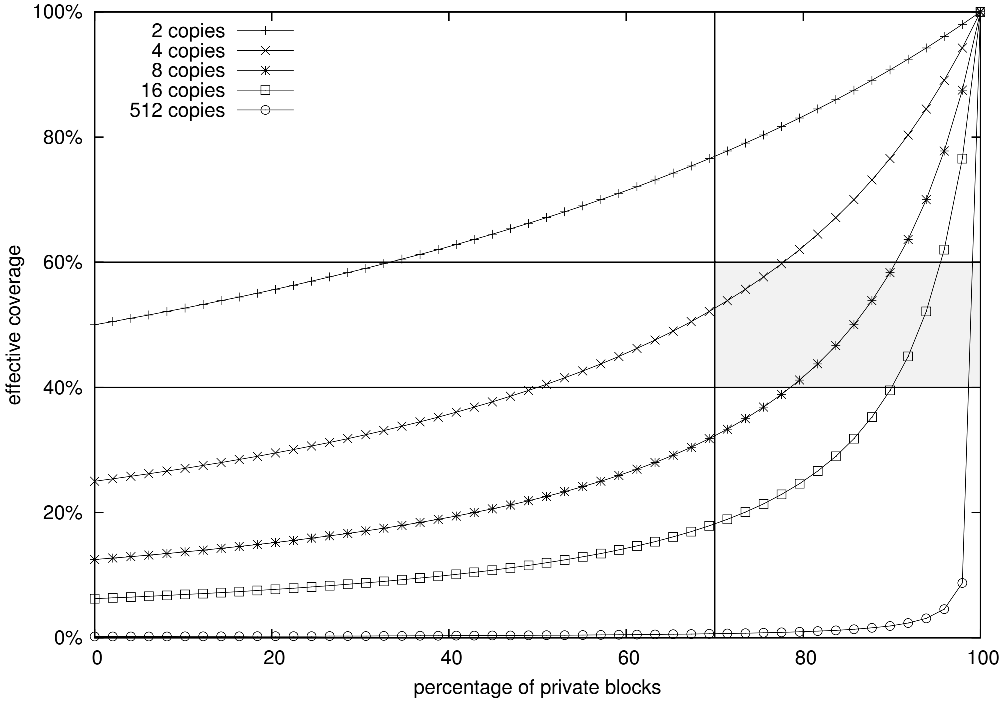


Figure 5.9: Effective coverage depending on memory block sharing characteristics.

5.3.4.1 Effective Coverage Analysis in Typical Scenarios

Effective coverages are completely workload-dependent, and they vary with the number of unique memory blocks stored in the private caches of the chip. Each of these memory blocks requires (at least) one directory entry to track all the copies of the block stored in the private caches (it might require more entries in composable schemes such as SCI). In general, the instantaneous effective coverage in a CMP can be characterized by means of the percentage of blocks that are private (only one copy of the block exists in the private caches), and the average number of copies of each of the remaining (shared) blocks (which by definition will be two or more). The higher the percentage of shared blocks and number of copies, the smaller the effective coverage. Smaller coverages will benefit ICCI, as its storage overhead is proportional to the effective coverage.

When calculating effective coverages, it is important to distinguish between memory blocks (of which several copies may exist) and private-cache blocks (several of which may be copies of the same memory block). This differentiation results in two possible methodologies when characterizing the private/shared

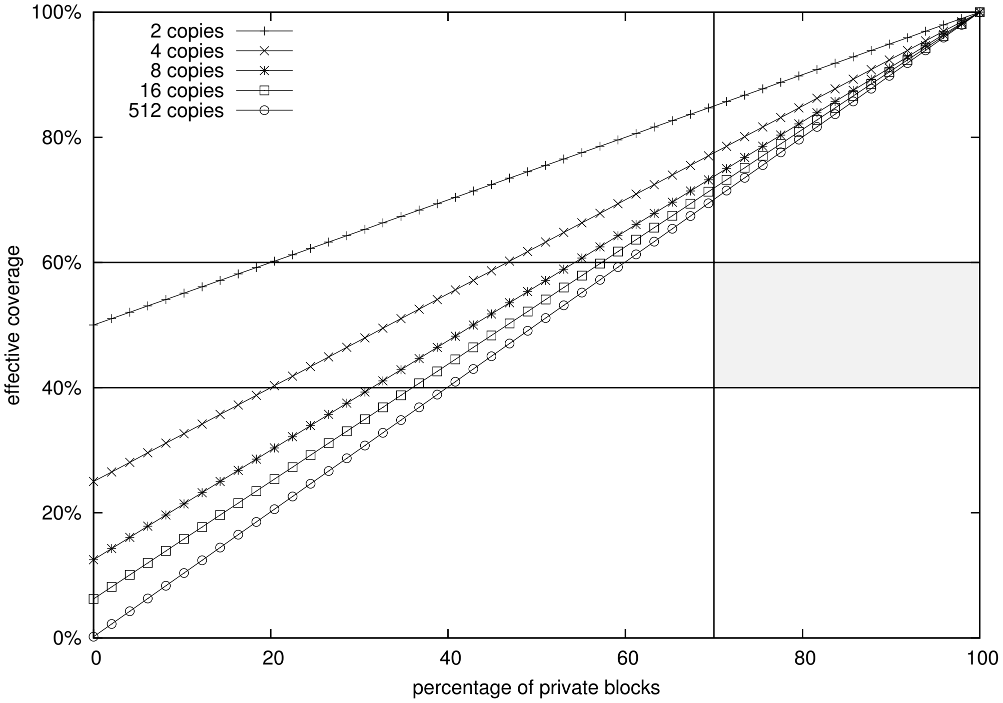


Figure 5.10: Effective coverage depending on private-cache block sharing characteristics.

block percentages, depending on whether the percentages refer to memory blocks or to private-cache blocks. Figures 5.9 and 5.10 show the effective coverages resulting when using each of these two alternative methodologies, respectively, for varying average number of copies for shared blocks. Even though the usual way to calculate these percentages is by considering memory blocks [6,16,45,51,71,141], we also show the calculations when considering private-cache blocks for completeness and to avoid ambiguity, as our graphical analysis (using conservative sharing values) will provide similar insights regardless of the methodology used.

To illustrate the differences between these methodologies, consider that, when counting private-cache blocks, each shared block is counted repeatedly, once for each copy of the block. With this methodology, the same runtime scenario would yield much smaller percentages of private blocks than if counting memory blocks (where shared blocks are counted just once), altering many of the values reported in the literature. Conversely, the same percentage of private blocks has different meanings depending on the methodology used. When counting private-cache blocks, 90% of private blocks involve an effective coverage over

90%. This is not the case when counting memory blocks, as the remaining 10% of shared blocks may have many copies, resulting in a very small effective coverage. Take a combination of 90% of private blocks and 16 copies for shared blocks. When considering private-cache blocks, the effective coverage is over 90%. When considering memory blocks, the effective coverage is under 40%. Even though using the same percentages and number of copies, the two methodologies result in two very different execution scenarios.

It is generally agreed that, especially in scientific applications, just a small percentage of memory blocks are shared, part of which are typically widely shared. For many workloads, it has been empirically observed that a majority of memory blocks are private to particular cores, with typical percentages ranging between 70% and 100% [17,39,72,151]. It has also been reported that on average just around 16% of the memory blocks stored in private caches are shared in the PARSEC benchmark suite [154]. Nevertheless, applications exist with a myriad of footprints, including high percentages of shared blocks. It is also widely agreed that this is typically the case of commercial workloads (in contrast to scientific workloads), as reflected by the following percentages of shared memory blocks that have been reported for some commercial workloads: 49% [16], 34% [71], 50% [17], 58% [127] in apache; 49% [16], 48% [17], 62% [127] in oltp; 29% [16], 38% [71] in zeus.

In addition, effective coverages around 40–60% are commonly reported in the literature [51,159] for scientific applications, some of them as low as 20% [159]. These small coverages have been leveraged by using complex hashing functions to reduce the amount of directory-induced invalidations with low physical coverages (close to or as low as 100%) [51,159]. The combination of typical percentages (private block percentages over 70% and effective coverages between 40–60%) results in the highlighted rectangle on Figure 5.9. We can see in the figure that those typical percentages are compatible with many possible values of average number of copies for shared blocks. As a curiosity, note that these typical private block percentages (over 70%) and effective coverages (many under 70%) do not match with the values in Figure 5.10 calculated counting private-cache blocks (i.e., the highlighted rectangle is empty). If the same execution scenarios of the highlighted rectangle of Figure 5.9 were plotted in Figure 5.10, the percentage of private blocks would be obviously smaller, as pointed out earlier.

In general, as the core count rises, more opportunities for (wide) sharing arise [109]. This observation has been taken into account in several recent works that evaluate the effects of data sharing in multiprocessor design [107,182]. In particular, as data sharing rises along with core count, it has increasingly critical impact on the miss rate differences between private caches that replicate shared data and shared caches that just store one copy of each memory block. Likewise, increases in data sharing make effective coverages decrease notably, as one directory entry is enough to track all the copies of a memory block. Notice that no sharing is possible in a chip containing a single core, which would always present a 100% effective coverage; for 2 cores, sharing may exist and as a result the effective coverage can range between 50% and 100%; as soon as we move up to 16 cores or 64 cores, effective coverages can be as low as 6.3% and 1.6%, respectively. Note in Figure 5.9 that, even assuming that 70–90% of blocks are private, if shared blocks have 16 copies on average, the effective coverage already ranges between 20% and 40%. If private data is less than 70% (as is typical in commercial workloads) or if the average number of copies of shared blocks is higher (which can be common in any parallel application with high levels of sharing), the effective coverage will be much lower. With such small effective coverages, ICCI will take up very little per-core storage space, while the overhead of separate directories is insensitive to any of these circumstances.

In fact, even small numbers of widely shared blocks result in low effective coverages. One of the elements that can reduce effective coverages notably is program code widely shared in parallel applications. In Figure 5.9 we show, just as a very optimistic reference, the coverage required when shared data is widely shared by 512 cores on average. In this case, even with 90% of blocks being private, the effective coverage is just 1.9%.

In Figure 5.11, we show a survey of effective coverages, including a considerable amount of empirical effective coverages found in the literature. In this figure, we show in semi-logarithmic scale the highest possible coverage (i.e., 100%, one directory entry per private-cache entry), that remains constant regardless of core count. We also show the minimum possible coverage, which takes place when each directory entry tracks as many sharers as there are cores in the chip (e.g., under 1% for 128 cores, with each directory entry tracking 128 sharers). In addition, the following relevant information is plotted:

- **Optimistic coverages.** Oh et al. [142] explore the optimal area breakdown between caches and cores for CMPs. In their analytical model, they optimistically assume that half of the cores share every memory block on average, based on their experience. In this model, the effective coverage at 512 cores would be just 0.4%. With this value, ICCI would take up just one twelfth as much area as a duplicate-tag directory. In fact, ICCI would take up less area than duplicate-tag directories with just 40 cores. Even though

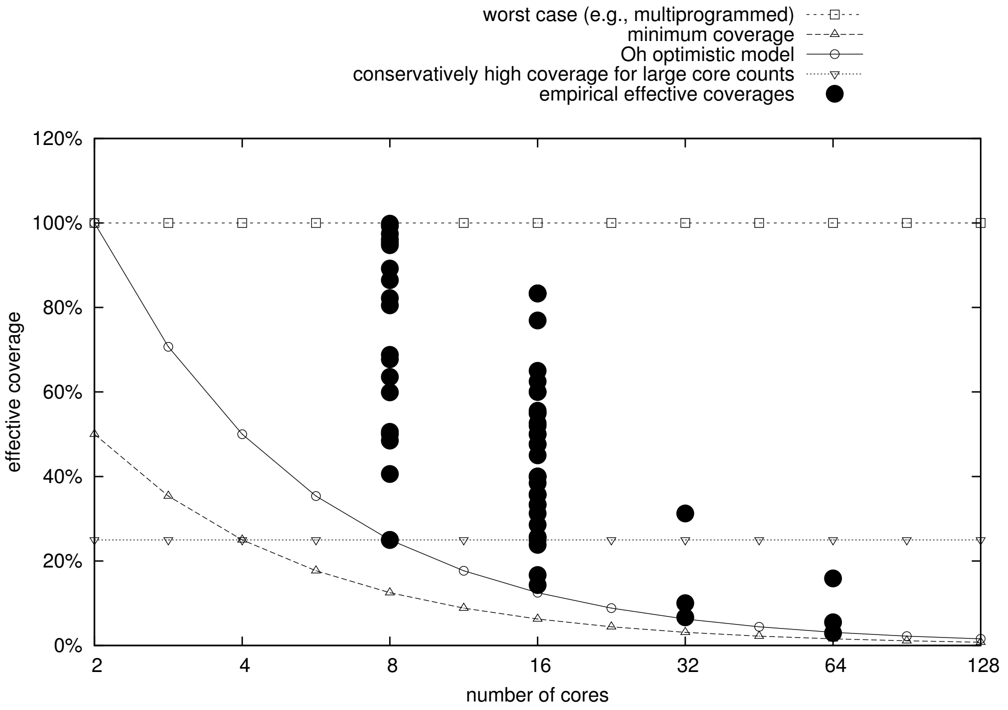


Figure 5.11: Survey of effective coverages.

possible (and confirmed by some empirical values, as we will see next), we consider this model too optimistic to be considered of general applicability.

- **Empirical coverages.** We have reviewed the literature and collected data from previous studies where the necessary information to calculate effective coverages was available [6,16,45,51,71,141]. All these data (61 values in total) are plotted in Figure 5.11, showing the effective coverages observed in a wide range of scientific and commercial workloads, from 8 to 64 cores. For eight cores, effective coverages near 100% are common. Nevertheless, effective coverages as low as 25% have been reported [6] with just 8 cores. Note that this value is surprisingly close to the minimum possible effective coverage for 8 cores, 12.5%, and matches the optimistic analytical model of Oh et al. [142] previously discussed. As the core count increases, a decreasing trend can be observed, with effective coverages down to 3% for 64 cores. Oh [141] provides empirical data on data sharing in the PARSEC benchmark suite. This data indicates that, for 64 cores, the highest effective coverage in these benchmarks is below 16%. Note that some empirical effective coverages are even smaller than the values predicted by Oh's optimistic model.

- **Conservative estimated coverages.** Rather than optimistically evaluating ICCI, we assume conservatively high effective coverages to compare ICCI against other directory schemes, in order to prevent an overestimation of its low-overhead features. We consider reasonable that typical effective coverages for parallel applications running on hundreds of cores can go up to 25% (even with hundreds of potential sharers). This is a conservative value taking into account that the empirical results previously discussed suggest smaller coverages and the fact that effective coverages for such core counts can potentially be well under 1%. In addition, we must never lose sight of multiprogramming and virtualization, which may potentially raise effective coverages up to 100%, regardless of the core count, by running independent applications in all cores (Section 5.3.4.2).

Separate directories, whose sizes are fixed at design time, cannot take advantage of small effective coverages. They would just result in many unused directory entries. An option would be to reduce the physical coverage of these directories at design time, counting on the prevalence of small effective coverages. However, this would be a very risky practice, as applications with memory footprints dominated by private data would raise the effective coverage over the directory physical coverage, incurring huge amounts of directory-induced invalidations, and performance would drop dramatically. Even in applications with low effective coverages, specific phases of execution may raise the effective coverage temporarily and ruin performance. On the other hand, ICCI suffers none of these problems. Should the effective coverage go up, even to its highest (100%), ICCI would seamlessly allocate as many directory entries as necessary. Should the effective coverage be very low, ICCI would just take up the minimum required amount of directory entries and let almost all the storage resources be used for storing data.

5.3.4.2 Boosting the Scalability of ICCI

ICCI's overhead will typically be small for parallel applications with low effective coverages. However, scenarios such as multiprogrammed or virtualized workloads, in which independent applications run in different cores with potentially no data sharing at all, can make effective coverages be close to 100%. These represent ICCI's worst-case scenario.

We can use complementary techniques to improve the situation on these worst-case scenarios. Mechanisms that detect different kinds of memory blocks (e.g., private or shared) at runtime have been proposed with a number of purposes [39, 76, 113]. In particular, it is interesting to consider a recently proposed mechanism that aims at reducing directory-induced invalidations on sparse directories [39]. These invalidations are not an issue in ICCI, but the same mechanism is very suitable for increasing ICCI efficiency in a completely different way.

First, we explain what the basics of this mechanism are. At a memory-page granularity, blocks are initially considered to be private to the core that first accesses them. The identifier of the accessing core and the private status of the page are stored in the page table of the process. No information for the blocks of private pages is stored in the directory. When another core attempts an access to a block belonging to a private page, it first receives the page table entry for the memory page (to carry out the virtual-to-physical address translation for the block) that also contains the private status of the page. This means that copies of blocks of that page may exist in the private-cache of the previous core, but no directory information exists for them. This triggers a procedure that creates directory entries for the former private blocks, which become shared, and retrieves the memory block from the private cache of the previous core if necessary. The page table entry is modified to indicate the new (shared) status of the page. This proposal has been reported to attain an average effectiveness over 75%, which means that it is able to deactivate the use of directory entries for more than 75% of private blocks. The remaining private blocks (less than 25%) belong to shared pages, and the page-level mechanism is unable to detect them. The effect of this mechanism was referred to as *deactivating coherence* for private blocks.

Originally, this mechanism was used to alleviate the pressure on sparse directories by reducing the number of blocks contending for the entries of the directory. This, in turn, reduced the amount of conflicts and evictions in the directory, preventing directory-induced invalidations and increasing system performance drastically.

Nothing prevents this mechanism from being directly applicable in combination with ICCI. Our observation is that, when coherence is deactivated for private blocks in a system implementing ICCI, no directory entries are allocated for them in the LLC. In contrast to sparse directories, where coherence deactivation just causes entries to be unused in the fixed-size directory, in ICCI this causes LLC entries to be used by data blocks instead of sharing codes. In both cases, the principle is the same: the effective coverage goes down; however, the side-effects are

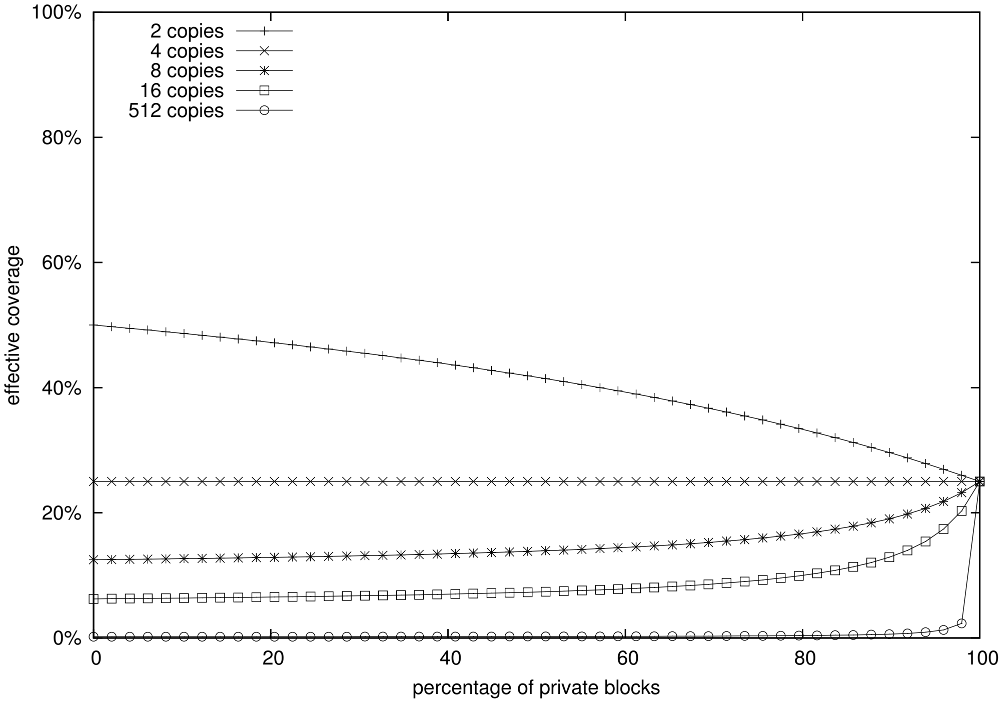


Figure 5.12: Effective coverages with 75%-effectiveness private-block coherence deactivation. Memory blocks considered.

very different. Note that this mechanism will also reduce the effective coverages of parallel applications.

With this mechanism, the worst scenario for ICCI, 100% effective coverage, becomes a potentially perfect scenario. For instance, if many different single-threaded applications are running on the CMP, with all their memory pages being private (with a different table page per process), this mechanism should easily deactivate coherence for all private-cache blocks, and ICCI's dynamic allocation of directory entries would not allocate any LLC entries for directory information (instead of one entry per private-cache block), introducing no area overhead for coherence in practice (except for shared OS data and code). No other coherence scheme provides such adaptability to runtime characteristics.

Figures 5.12 and 5.13 show the effective coverages resulting when applying this mechanism, assuming an effectiveness of 75% (smaller than the reported average effectiveness for the mechanism). Note how, for common percentages of private blocks (over 70%), the required coverage is always under 40% in the worst case (just 2 copies per shared block), and can be easily under 20% as soon as

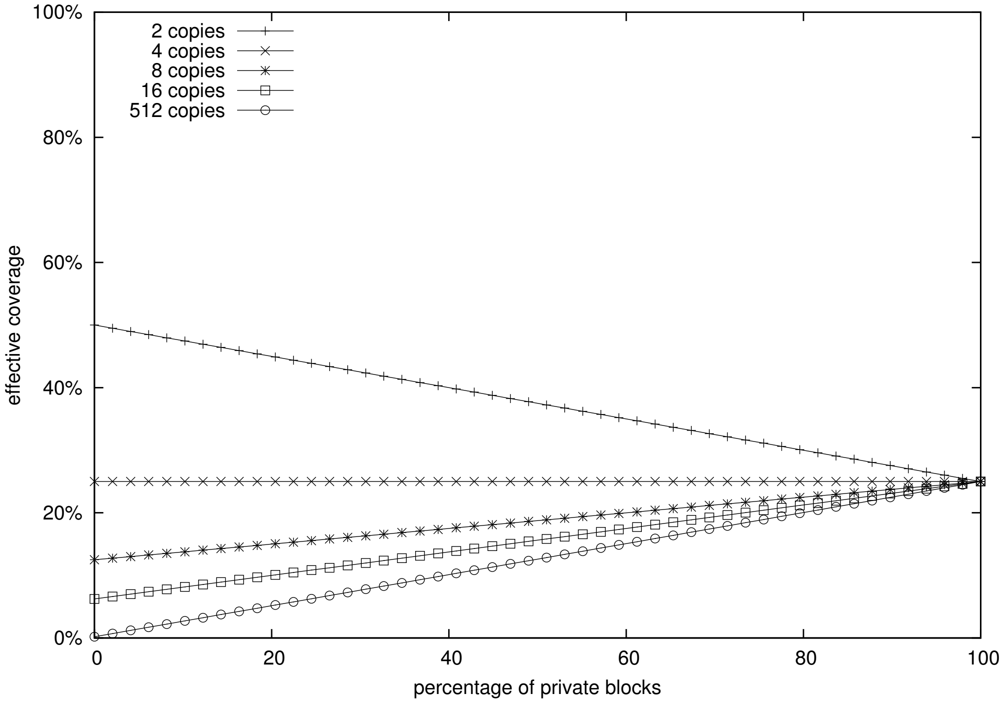


Figure 5.13: Effective coverages with 75%-effectiveness private-block coherence deactivation. Private-cache blocks considered.

shared memory blocks have more than 4 copies on average. The typical cases that were highlighted in a rectangle in Figure 5.9 always result in effective coverages equal to or below 25% after deactivating coherence for private blocks with 75% effectiveness. These results lead us to assume 25% again as a conservatively high upper bound for effective coverages with large core counts (the same percentage assumed for parallel applications in Section 5.3.4.1). We also choose this value because it corresponds to four copies per shared block regardless of the amount of private blocks and the methodology used to count blocks (see the flat line of Figures 5.12 and 5.13) at 25% effective coverage, which is a conservative scenario for large core counts. In conclusion, the possibility of deactivating coherence for private data makes the scalability of ICCI benefit from low effective coverages regardless of the private/shared footprint of the particular workload in execution.

As discussed for wide data sharing scenarios, separate directories could be scaled down to physical coverages under 100%, in the hope that the coherence-deactivation mechanism will always attain low effective coverages. However, this mechanism does not give any guarantees on the deactivation of a single block

(Figures 5.9 and 5.10 still represent the case in which the mechanism has an effectiveness of 0% in addition to the case in which the mechanism is not used). As explained previously, using low physical coverages in the fixed-size directory would be very risky (and very-low ones, matching expected effective coverages under 25%, are out of the question). On the other hand, ICCI dynamically benefits from low effective coverages at runtime, potentially requiring 0% of storage resources for cache coherence.

The only advantage remaining in favor of using a fixed-size directory is their smaller entry size for low core counts. This may make ICCI take up more storage resources even when benefiting from small effective coverages. However, as the core count rises and sharing code size approaches LLC entry size, fixed-size directories lose this advantage and fail as a scalable scheme, while effective coverages go down benefiting ICCI's scalability.

Once sharing code size surpasses LLC entry size, composable sharing codes (similar to SCD) specific for ICCI can be used, with the possibility to store many sharers and point to other entries that make up the sharing code at the same time thanks to the large LLC entry size. We find this possibility especially appealing. For instance, SCD's 2-level hierarchy for 1024 cores can be built with just 3 LLC entries in ICCI, instead of the original 33 SCD cache entries. This means a much higher entry-efficiency and faster lookups than in SCD. 512-bit LLC entries can support a hypothetical 2-level ICCI-SCD coherent system containing 256K cores, with no dedicated storage overhead for coherence. In addition, 512-bit entries can store up to 28 pointers (to any of the 256K cores), before needing to use multiple entries, ensuring a high entry-efficiency. Note that when several LLC entries are needed for tracking one memory block, many private-cache blocks share each of these entries, resulting in a reduction of the effective coverage. Increasing the cache line size to 128 bytes would potentially enable cache coherence for a 1M-core machine. Also, the number of levels in the hierarchy can be increased to support more cores. In addition, ICCI's large directory (with potential effective coverage equal to the percentage ratio of the number of LLC entries to the number of private-cache entries) prevents conflicts and directory-induced invalidations naturally, removing altogether the need to use ZCaches to simulate large associativity in small coverage directories.

In addition, techniques similar to Amoeba Caches [108] can be used to enable different entry sizes in the LLC, to accommodate (small) directory entries and (large) cache entries. However, as the sharing code size approaches the LLC entry size, this results in an unnecessary complication. Nevertheless, taking this path one step further may be interesting, as different sharing codes could be

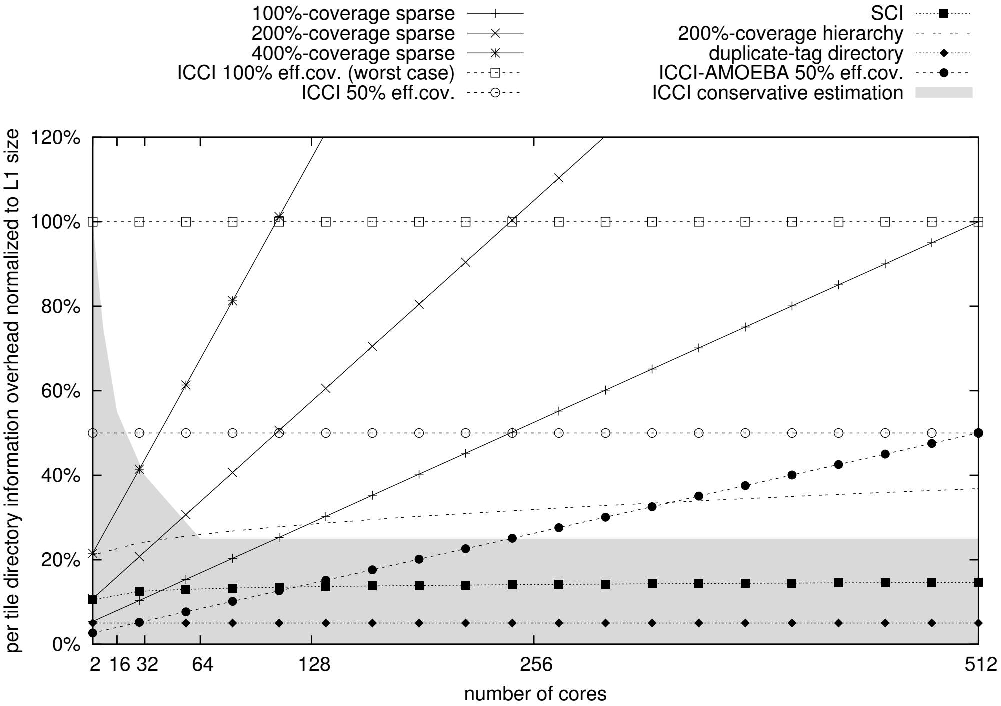


Figure 5.14: Cache coherence storage overhead of several schemes depending on core count (from 2 to 512 cores).

used to minimize the total space taken up by coherence information in ICCI (e.g., pointers for few sharers or a bit vector relative to an area for a block shared in that area of the chip).

5.3.4.3 ICCI Compared to Other Coherence Schemes

After the discussion on typical coverages, now we can put ICCI's storage overhead into perspective with other coherence schemes. Figure 5.14 shows the overhead of several proposals, ranging from 2 to 512 cores. The overhead introduced by these coherence schemes is measured as the percentage of storage space used for coherence information relative to the aggregate private-cache capacity. This figure shows sparse directories of several coverages (solid lines), ICCI for several effective coverages, a hierarchical directory (with two levels of sharing information and a 200% coverage in each level), SCI and a duplicate-tag directory. We assume an SCI version adapted to CMPs, in which a 200%-coverage directory cache storing pointers is NUCA-distributed and each L1 cache entry contains two pointers to create the double-linked list characteristic of SCI. In this figure, ICCI uses the same encoding for sharers as the sparse directory (full-map bit-vector).

The characteristics of ICCI are a bit unusual. Its overhead depends on effective coverage rather than on core count. It remains constant for a given effective coverage because the number and size of the LLC entries used as directory entries is the same regardless of the number of cores. In addition, the entry size of the LLC is the same as the size of a private-cache entry, making the overhead of ICCI on private-cache capacity approximately equal to the value of the effective coverage. For instance, the worst-case effective coverage of 100%, with one LLC entry tracking each private-cache block, results in (approximately) 100% storage overhead on L1 cache capacity regardless of the number of cores. The only difference introduced by changing the core count is that, as the core count and the number of banks in a NUCA cache increase, more address bits are used to select the home LLC bank, and the number of remaining bits used in the tags of the LLC goes down. This is such a subtle difference that its impact on the overhead of ICCI cannot be appreciated in the graph.

Note that, contrary to traditional directories, ICCI will typically take up less storage per tile for directory information as core count increases. Higher core counts can potentially experience higher data sharing, generating smaller effective coverages and making the area overhead of ICCI go down as the core count goes up.

The shadowed area of Figure 5.14 represents the expected range of overheads for ICCI, assuming the conservatively high upper bound for effective coverages for large core counts (25%) that was calculated in Sections 5.3.4.1 and 5.3.4.2. Also, an estimated overhead for ICCI in combination with Amoeba Caches is shown, for a 50% effective coverage, which beats the sparse directory easily even with small core counts. However, the higher the core count, the less savings this alternative provides and the less sense it makes to use it.

Several facts stand out in this graph:

- A 200%-coverage sparse directory uses twice as many resources as ICCI in its worst-case scenario (i.e., 100% effective coverage) for 512 cores. ICCI's area overhead for typical effective coverages range between 12.5% and 0% that of the sparse directory. The sparse directory uses more resources than ICCI with 25% effective coverage as soon as the core count rises over 50 cores, and its overhead keeps rising with core count while in ICCI it goes down as more cores increase data sharing.

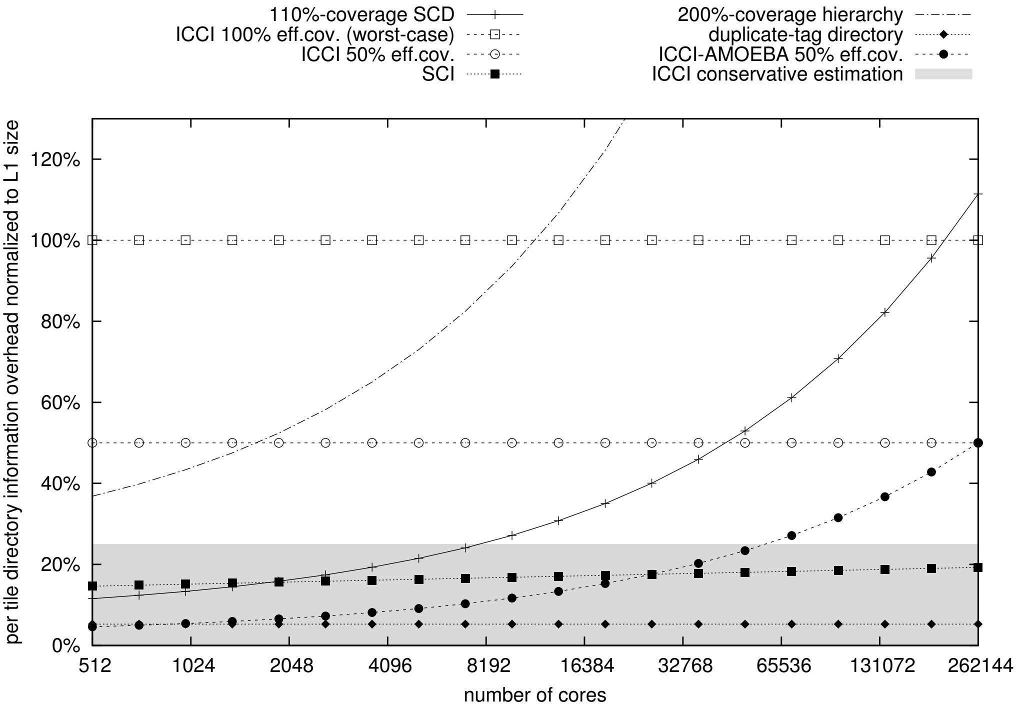


Figure 5.15: Cache coherence storage overhead of several schemes depending on core count (from 512 to 256 K cores).

- The 100%-coverage sparse directory uses as many resources as ICCI's worst-case with 512 cores and four times as many as ICCI with 25% effective coverage. In addition, a sparse directory with just 100% coverage would produce unacceptably large numbers of directory-induced invalidations due to conflicts.

- ICCI provides reasonable overhead for up to 512 cores based on expected effective coverages, without dramatic changes and without the complexity of more intricate alternatives such as hierarchical protocols. Note that with just 64 cores and a 25% effective coverage, ICCI already takes fewer resources than the hierarchical directory.

- ICCI has lower overhead than SCI when the effective coverage is below 16% and lower than duplicate-tag directories when the effective coverage is below 5%. Some empirical effective coverages observed are under these values, even for low core counts (see Section 5.3.4.1).

Figure 5.15 shows the storage overhead on the L1 caches for core counts between 512 and 256 K, in semi-logarithmic scale. In this case, ICCI uses the same composable codification as SCD. When ICCI uses composable sharing codes (taking up several entries when many sharers exist, like in SCD), the relation between effective coverage and overhead may vary ever so slightly (because the one-to-one relationship between directory entries and memory blocks is broken), although they remain roughly equivalent. Nevertheless, this has no effect on the fixed coverages shown in the figures (in which each LLC entry used for directory tracks one or two private-cache blocks) and on the conservative upper bound for effective coverages. In this case, sparse directories are not shown, as their overhead is well out the charts (e.g., over 50000% for 256K cores in a 200%-coverage sparse directory).

Interesting results are the following:

- SCD goes over 100% overhead eventually, while ICCI with SCD's sharing code remains under 25%. In addition, ICCI does not require the use of ZCaches, as explained in Section 5.3.4.2.

- Again, due to the particular characteristics of ICCI, it has lower overhead than SCI when the effective coverage is below 16% and lower overhead than duplicate-tag directories when the effective coverage is below 5%.

As the size of the sharing code approaches the LLC entry size with increasing core counts, ICCI has obviously superior scalability properties than a separate directory using the same sharing code (e.g., full-map in a sparse directory for up to 512 cores and a composable hierarchical code like SCD's for up to 256 K cores).

We conclude that ICCI's effective-coverage-dependent overhead is comparable to very scalable schemes in terms of area, such as SCI.

5.4 Related Work

Novel coherence schemes appear periodically in the literature, and the complexity of the most recent ones shows that it is becoming increasingly difficult to improve the scalability of cache coherence.

For instance, the Tagless Coherence Directory (TL) [180] uses multiple-hash bloom filters to store directory information. In essence, TL works as an inexact duplicate-tag directory (inexactitude due to bloom filter aliasing, which creates spurious invalidation messages). Ideally, TL has constant per-core overhead. In practice, the bloom filter size has to be increased with the core count to prevent excessive levels of aliasing, in a trade-off between extra network traffic and area overhead. In addition, although more energy-efficient than a duplicate-tag directory, TL is less energy-efficient than ICCI. In ICCI, an LLC lookup is enough to find the block or the sharing vector. TL requires an additional directory lookup whose energy consumption is proportional to the number of cores. In our 512-core CMP, a TL access requires looking up 512 multiple-hash bloom filters in parallel to generate the 512-bit sharing vector. Overall, TL introduces the difficulty of managing bloom filters in hardware, extra resources for the directory, and the inefficiency of spurious invalidation messages compared to ICCI.

SPACE [185] is based on the observation that many cache blocks have the same or similar sharing patterns. SPACE stores these sharing patterns in a table. The directory cache stores pointers to positions of the pattern table, one pointer per tracked block, with many directory entries pointing to the same patterns. As long as the pointer directory cache dominates the overhead, SPACE can scale up gracefully with core count. However, as the number of cores grows, the pattern table starts to dominate (note that the table is distributed and patterns need be repeated at every tile) resulting in a per-core overhead proportional to the number of cores (i.e., SPACE incurs the same inefficiencies as any fixed-size directory). SPACE assumes that few different sharing patterns exist at any given time; hence a small sharing pattern table is needed, resulting in a smaller overhead than an ordinary sparse directory in any case. However, as the number of cores grows, the number of possible sharing patterns increases exponentially; hence the possibilities of pattern repetition diminish. SPACE also introduces the complexity of managing the sharing pattern table, which requires non-trivial actions such as pattern coalescing. ICCI suffers none of these problems.

SPATL [186] combines both the Tagless Coherence Directory and SPACE, storing the pointers to the sharing pattern table inexactly in bloom filters, reducing the overhead further at the cost of the aggregate complexity of both proposals. Unfortunately, for large core counts, SPATL faces the same scalability problems as SPACE due to the size of the pattern table.

The Cuckoo Directory [51] uses a different hash function per directory way so as to prevent directory conflicts. This reduces the need for overprovisioning cache directories, but does not solve any of the inefficiencies of using fixed-size directories and does not change their per-core overhead, which remains non-scalable. The reported absolute area savings for Cuckoo Directory are so high because Cuckoo Directories with coverages near 100% are compared against huge 800%-coverage sparse directories. Note that we assumed 200%-coverage sparse directories throughout this chapter. Directory-induced invalidations are not an issue in ICCI due to the large size of the LLC, making complex hash procedures such as Cuckoo hashing unnecessary.

The SGI® UV2 [168] uses directory-based cache coherence to maintain 512-processor-socket coherent domains. The full directory is stored in DRAM, typically consuming approximately 3% of the 64 TB memory space, and an on-chip directory cache allows for fast access to information about reused addresses. Similarly, WayPoint [97] uses small, low-associativity directory caches. Evicted directory entries are inserted in main memory to prevent costly directory-induced invalidations. ICCI allows for 512-processor coherence domains without the need for the slow-access DRAM directory or the directory cache used by the SGI UV2 and WayPoint.

We have already discussed the Scalable Coherence Directory [159] and used it to compare ICCI effectiveness.

5.5 Conclusions

In this chapter, we have introduced ICCI, a new cache organization that leverages shared cache resources and flat coherence protocols to provide inexpensive hardware cache coherence for large core counts (e.g., 512), without degrading the performance and energy consumption of the system as other proposals do (e.g., coarse bit vectors, SCI) and without the need of complex cache structures (like SCD's ZCaches). Simple changes in the system are needed to implement ICCI with respect to a traditional full-map directory. ICCI does not introduce any dedicated storage overhead, yet it provides large storage space for coherence information. ICCI takes up entries of the LLC as directory entries. ICCI incurs a negligible number of directory-induced invalidations and outperforms complex state-of-the-art proposals such as SCD, especially in terms of energy. Moreover, ICCI can be used in combination with more elaborated sharing codes to apply it to extremely large core counts. By combining ICCI and SCD, we can provide hardware cache coherence for massively parallel machines at no extra chip area cost. This can be done by storing SCD's hierarchical entries in the LLC and using ICCI's operation. ICCI removes the need to orchestrate two different array structures for data and directory information and the use of complex coherence protocols (e.g., hierarchical, list based or tree based).

We have carried out an analytical survey of the characteristics of workloads, concluding that low effective coverages typical at runtime ensure high scalability for ICCI in terms of storage taken up for directory information in the LLC. In the presence of data sharing, effective coverages typically below 25% make ICCI take up few directory entries and add little overhead. In the absence of data sharing, deactivating coherence for private blocks also enables low effective coverages (typically under 25%), making ICCI take up few directory entries under any circumstances. In comparison, a fixed-size directory always takes up the same amount of entries, determined at design time to fit worst-case scenarios, and leaves them unused if there is wide data sharing or if coherence for private blocks is deactivated.

ICCI's logical directory size (number of sets and associativity) is huge compared to dedicated storage directories. Directory-induced invalidations are never an issue in ICCI. This prevents the need for complex caches (e.g., ZCaches) or hash procedures (e.g., Cuckoo hashing) to emulate large associativity in small fixed-coverage separate directories to prevent directory-induced invalidations.

Finally, contrary to any other directory scheme, ICCI's resource usage for directory information typically decreases as the number of cores rises, because more opportunities for data sharing appear, reducing the effective coverage and the number of allocated directory entries. In addition, reported effective coverages make ICCI take up less area for directory information than SCI or duplicate-tag directories with as few as 64 cores.

Chapter 6

Dynamic Management Policies for Exploiting Hybrid Photonic-Electronic NoCs

Nanophotonics promises to solve the scalability problems of current electrical interconnects thanks to its low sensitivity to distance in terms of latency and energy consumption. While this technology reaches its maturity, hybrid photonic-electronic networks have started to be a viable alternative to purely electrical networks. Ideally, these hybrid networks should exploit the best features of each technology. For example, an ordinary electrical mesh and a ring-based photonic network can cooperate to minimize their overall latency and energy consumption. However, we currently lack mechanisms to do this efficiently. In this chapter, we present novel fine-grain policies to manage photonic resources in a tiled-CMP scenario. Our policies are dynamic and base their decisions on parameters such as message size, ring availability and distance between endpoints, at the message level. The resulting network behavior is also fairer to all cores, reducing processor idle time thanks to faster thread synchronization. After designing and evaluating a wide range of policies with different features, we conclude that the most elaborate ones reduce the overall network latency by 50%, execution time by 36% and the energy consumption of the network by 52% on average, in a 16-core CMP for the PARSEC benchmark suite, when compared to the same CMP without the photonic ring. We also show that larger hybrid networks with 64 endpoints for 256-core CMPs, based on Corona and Firefly designs, also achieve far superior throughput and lower latency if managed by an appropriate policy that exploits both photonics and electronics.

6.1 Background

To be able to keep pace with Moore's Law under the current power-constrained scenario, the latest generations of most microprocessors have adopted an on-chip multi-core architecture where the last-level cache (LLC) is typically distributed across tiles [100]. This configuration enables scalability, but while each core can directly access the portion of cache in its own tile, it needs to use an interconnection network to access the cache resources in other tiles. The tiled-CMP design paradigm (described in Section 1.4) is devised to run complex and heterogeneous multi-threaded applications, which require efficient communication and synchronization between threads within the chip. This leads to the need for efficient and effective on-chip interconnection mechanisms such as high-performance and structured network-on-chips (NoCs) for interconnecting the tiles (each one including typically cores and cache resources).

The execution time of applications is becoming more and more affected by network traffic, and in particular by the average distance traversed to retrieve the data from the correct LLC tile in the chip (i.e., the number of network hops). As the core count increases, the number of retransmissions that messages suffer in the electrical network also increases, compromising the scalability of future chip multiprocessors (see Section 1.3 for further details). In addition, data transmission through the on-chip interconnect is starting to account for most of the energy consumption of a chip, and this scenario is expected to get worse in future CMP systems [27, 122]. This problem must be addressed to continue increasing the performance of future chips within a reasonable power budget.

Advances in silicon photonics have enabled the integration of optical interconnects inside silicon chips [68,90]. This disruptive technology provides low-energy fast data transmission across the whole chip regardless of the distance, and can be a solution to the scalability problems of NoCs. For instance, transmitting information between two opposite corners of a 8×8 electronic mesh at 4 GHz requires traversing fifteen routers and fourteen inter-tile links, taking up tens of processor cycles (also at 4 GHz). However, traversing the same distance in a photonic waveguide[1] can take as little as two processor cycles, needs no retransmissions, and uses significantly less energy in the process.

[1] about 15 ps/mm (group velocity of light into silicon)

Investigation on the maximum benefits achievable from simple optical topologies (e.g., rings) is strategic because, for relatively short-term commercial solutions, the use of a simple-topology photonic network, possibly 3D-stacked, can be an interesting design choice. In addition, a number of ring-based photonic topologies have been proposed as scalable networks for connecting large numbers of cores [145, 147, 173, 178] in the future.

Hybrid photonic-electronic NoCs attempt to make the most of both transmission technologies [10, 82, 114, 147]. The presence of two different transmission technologies (each with a different behavior with respect to the traffic injected) resembles heterogeneous electrical networks [35,54]. However, the characteristics of photonic networks change the rules of the game significantly: while in the classic heterogeneous network scenario the low-latency network was very power-hungry and should be used selectively for accelerating certain messages, in this new scenario the *ultra*-low-latency photonic network is the least power consuming network of the system, but it can suffer from long message queuing latencies due to serialization, reducing its potential benefits when its load increases. The latency and energy advantages of photonic networks (especially the simple ones that are more likely to be implementable soon) may be wasted if not carefully managed. Currently, there is a lack of adequate policies to carry out such careful management. Our purpose is to develop mechanisms to make effective use of any amount of optical resources, completely or partially shared by a number of cores in a CMP.

In this chapter, we explore novel dynamic policies aimed at making the best use of a photonic network that works in cooperation with an electrical mesh. To our knowledge, these are the first proposed ad-hoc management strategies that use real-time information for hybrid photonic-electronic NoCs at the message level. We present these policies in increasing order of complexity. We test them on a CMP equipped with a modest photonic ring, which is a likely representative of near-future CMPs. We evaluate these policies and analyze their different characteristics in terms of execution time and energy consumption, finding that elaborate policies are able to notably reduce the average execution time of applications (in particular for the PARSEC benchmark suite [21]) and the energy consumption of the network compared to simpler policies. Then, we test the policies on larger networks to prove their general applicability to hybrid photonic-electronic networks for long-term chips. Here, we also show that large throughput and latency benefits are achieved only if proper policies are employed.

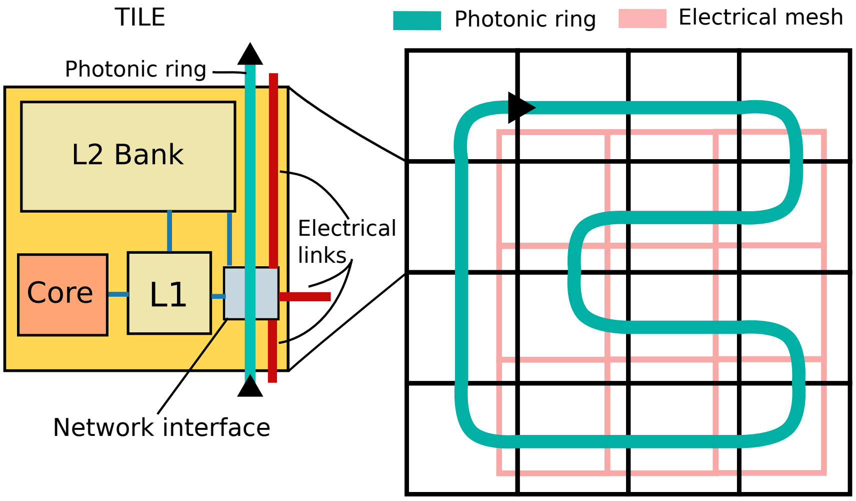


Figure 6.1: Hybrid photonic-electronic NoC on a 16-core tiled CMP. Every tile can read from or write in the photonic ring.

Next, we give the necessary background on photonics and on the network architectures selected to evaluate our policies.

6.1.1 Ring-based Photonic Networks

Due to their simplicity and ability to exploit fast photonic transmission (Section 2.2), photonic topologies such as rings [145,147,173,178] are likely to make their way into commercial machines before more complex photonic architectures like those that use photonic switches, *passive* or *active* (i.e., dynamically reconfigurable) [139,148,160,183], to emulate elaborate topologies (e.g., mesh, fully-connected).

In the case of passive switches, they always let one or more wavelengths go through the switch without turning, and divert one or more wavelengths to a different photonic output port [139,183]. This can enable a passive-routing interconnection by associating the wavelengths to origin-destination node pairs. These approaches need to employ a very large number of optical switches and incur many waveguide crossings which can introduce significant optical attenuations. On the other hand, they are able to deliver dedicated optical channels between each core pair. However, due to intrinsic technological limitations of optical switches, these passive networks cannot deliver great optical parallelism per source-destination pair (e.g., 1 bit per path for an 8 core CMP).

The use of active photonic switches [148, 160] needs a photonic-circuit establishment mechanism to configure the microring resonators through a supporting electrical network, before transmission. This is essentially due to the incapacity to implement routing within the optical domain. The circuit establishment overhead makes these photonic topologies less attractive for cache coherence based systems in which communications typically consist of quick transmissions of small packets (around 8 or 72 bytes typically [125]). In this case, the overhead of establishing the circuit in the mesh would largely outweigh the benefits of sending the packet through the photonic network.

Figure 6.1 shows a photonic ring as part of a hybrid photonic-electrical NoC for a 4×4 CMP. In a simple ring topology, photonic resources can be allocated in a number of ways. In the simplest form, each origin-destination pair is assigned a different set of wavelengths, preventing the need for arbitration. However, this limits the bandwidth for single transmissions too much. For example, assuming a 64-wavelength ring, an 8-core CMP can afford a single wavelength per origin-destination pair. Hence, every transmission is limited to just 1 bit per ring cycle, even if no other transmissions are taking place in parallel in the ring.

More flexible configurations are Single Writer Multiple Reader (SWMR) [103, 147], Multiple Writer Single Reader (MWSR) [173] and Multiple Writer Multiple Reader (MWMR) [145]. These provide more flexibility in exchange for some arbitration cost. In SWMR, each writer uses its own dedicated wavelengths, which can be read by any receiver. Before transmitting, a destination selection mechanism is used by the writer to make the adequate receiver turn on its photodetectors to read the data from the writer's wavelengths. In MWSR, each receiver reads different wavelengths and arbitration is needed on the writer's side (e.g., token channel, token slot [172]) to avoid collisions between writers. MWMR is the most flexible configuration, but it requires both arbitration in the writer's side and destination selection. In MWMR, a single transmission can use all the data wavelengths of the ring, maximizing the available bandwidth. This configuration requires that every destination can read and write every wavelength, which also increases the power consumption of the photonic ring due to the extra ring modulators and photodetectors necessary.

6.1.2 Case Study Photonic Networks

Here, we describe three notable networks using ring-based photonic topologies which have been recently proposed, one for each ring arbitration policy (MWMR, MWSR and SWMR). Table 6.1 describes the characteristics of these networks,

Table 6.1: Feature comparison between case study NoCs. Values with (*) are estimations based on the available information.

	System characteristics				
Solution	**Technology**	**Cores**	**Notes**		
FlexiShare	22 nm	64	Eight 512-bit channels		
Corona	16 nm	256	Photonics to DRAM		
Firefly	45 nm	256	Hybrid ph/el NoC		
	Optical features				
Solution	**N**$_{endp}$	**N**$_{waveg}$	**N**$_{micror.\ res.}$	**Access scheme**	**Phit size**
FlexiShare	8–32	130–138*	113–1052 K*	MWMR	512
Corona	64	388	1056 K	MWSR	256
Firefly	64	320*	130 K*	SWMR	256
	Electronic features				
Solution		**Concentr.**		**N**$_{links}$	
FlexiShare		8–2		0	
Corona		4		0	
Firefly		4		80	

with all of them using Dense Wavelength Division Multiplexing (DWDM) where up to 64 wavelengths are transmitted through a single waveguide. Later in this chapter, the effectiveness of our policies will be tested by exploiting hybrid NoCs based on these photonic networks.

6.1.2.1 FlexiShare (MWMR)

FlexiShare [145] is an MWMR photonic ring proposed for a 64-core CMP. It introduces token stream arbitration to increase network utilization. FlexiShare was evaluated by their authors with varying values of radix (8, 16 and 32, corresponding to the number of network endpoints accessible through the ring) and concentration (8, 4 and 2, corresponding to the number of cores sharing each network endpoint). Also, different numbers of channels, each with a 512-bit datawidth, were tested. Arbitrating these channels is not trivial [176], and FlexiShare assumed round-robin channel selection. The values shown in Table 6.1 assume eight channels.

6.1.2.2 Corona (MWSR)

Corona [173] is a 256-core CMP containing a ring-based MWSR photonic network to interconnect 64 four-core clusters. Each of the 64 endpoints receives data through a dedicated 256-bit datapath that comprises four photonic waveguides, and senders compete for the right to use the channel. The resources needed by Corona are more than those required by the 64-core FlexiShare designs. Corona solves the multiple-channel arbitration problem by using dedicated channels for receivers, at the cost of wasting bandwidth under unbalanced traffic.

6.1.2.3 Firefly (SWMR)

Firefly [147] is a hybrid photonic-electronic network using photonic rings also for 256-core CMPs. In addition to using a concentration of four cores per endpoint, like Corona, Firefly's design groups its 64 endpoints in eight clusters of eight endpoints each. An electrical mesh per cluster carries intra-cluster traffic, benefiting from the high bandwidth of electrical links in short-distance transmissions. An SWMR photonic ring is used for inter-cluster traffic to enable fast long-distance communication, with a dedicated 256-bit channel for each writer. This channel connects the writer to just one endpoint per cluster, saving photonic resources (microring resonators and photodetectors) compared to Corona, where each and every channel connects all 64 endpoints. This results in photonic rings being efficiently used for long-distance communication (inter-cluster) without suffering the burden of short-distance transmissions (intra-cluster) that would increase packet serialization. This design also enables the removal of electrical links between clusters, providing static energy and area savings compared to using a full mesh.

6.1.3 Arbitration and Pipelined Transmission

Several arbitration mechanisms are possible for the photonic rings just discussed. For MWMR and MWSR rings, we use a simple token-passing arbitration mechanism because of its simplicity and its fairness. Using this technique, an emitter reads the token wavelength, and when a light pulse is detected (and therefore destroyed), the token has been acquired. We allow each emitter to send one message and then the emitter has to inject the token again in the waveguide. The main problem of this simple token passing mechanism is the underutilization of the photonic ring that may appear under certain conditions such as when there is a single emitter that has to relinquish the token periodically and wait for its arrival after circling the ring before transmitting again, wasting potential transmission slots in the ring. However, results show that a reasonable utilization is achieved with the single token ring mechanism across the PARSEC benchmark suite. Nevertheless, more complex arbitration techniques can potentially give a marginal improvement on the utilization of the ring. In any case, evaluating different arbitration mechanisms is out of the scope of this chapter, and the proposed policies are equally applicable along with more complex arbitration mechanisms. For SWMR rings, this arbitration mechanism is not necessary, as each emitter uses a dedicated datapath.

For the sake of fairness in the evaluation, we assume the same pipelined packet transmission for all networks. Between token acquisition and data transmission, a lapse of three ring-cycles takes place in which the activation of the destination's photonic receivers is performed by means of a light pulse on four wavelengths (for 16 cores), indicating the identity of the destination. This is not strictly necessary for MWSR rings as the wavelengths used for the transmission are associated to just one destination, but we use it nonetheless because the destination receivers may be optionally turned off to save energy in the absence of transmissions. The token is released before the real transmission takes place to enable the potential use of all transmission slots. All of the latencies involved in the photonic ring operation were modeled in the tests.

6.2 Dynamic Management Policies for Hybrid NoCs

In this section, we present a series of novel policies to efficiently manage hybrid networks comprising a ring-based photonic sub-network and an electrical sub-network such as a mesh. These policies decide which sub-network to use for each message, basing their decisions on real time information available when sending each message that comprises one or more of the following parameters: message size, photonic ring availability, and distance between endpoints. Table 6.2 summarizes the policies.

6.2.1 SIZE: Message Size

Our first criterion to make use of the photonic sub-network is the size of the message to transmit. There are typically two different kinds of messages in a cache-coherent CMP: control messages and data messages. Control messages

Table 6.2: Summary of policies

Configuration	Messages on photonic ring	Messages on mesh
SIZE	control messages (8 bytes)	data messages (72 bytes)
AVAIL-x	token acquired within x cycles	other messages (x-cycle delay)
DDA-x	token acquired within $(l_m - l_p) \times 0.x$ cycles	other messages $((l_m - l_p) \times 0.x$ delay)
CDDA-x	control if token acquired within $(l_m - l_p) \times 0.x$ cycles data if token acquired within 2 cycles	other messages $((l_m - l_p) \times 0.x$-cycle delay for control, 2-cycle delay for data)
MTDDA-x-y	control if token acquired within $(l_m - l_p) \times 0.x$ cycles data if token acquired within $(l_m - l_p) \times 0.y$ cycles	other messages $((l_m - l_p) \times 0.x$-cycle delay for control, $(l_m - l_p) \times 0.y$-cycle delay for data)

l_m = idle mesh latency, l_p = idle photonic ring latency.

commonly have a size of 8 bytes, while data messages add to this size the data block (usually 64 bytes) resulting in a total of 72 bytes [125]. The transmission of a data message makes the photonic ring unavailable for a much longer time than a control message, potentially increasing the latency of other messages.

For example, our chosen near-future network (see Section 6.3.1) makes use of a high-performance waveguide with 64 data wavelengths that enable the transmission of up to 8 bytes per ring-cycle. Therefore, control message and data message transmissions take 1 and 9 ring-cycles, respectively. In the span of time used to transmit a data message, nine short messages (one per ring-cycle) could be sent, greatly benefiting from the low latency of the ring. In other words, sending long data messages increases the chances of suffering long queuing times, due to serialization of messages, and decreases the opportunities to accelerate many other messages.

In general, short messages account for just a small percentage of the overall traffic in bytes due to their small size, although they account for most of the messages injected in the network (e.g., request, invalidation and acknowledgement messages). Thus, their acceleration provides a large performance gain with small bandwidth usage, and their small size reduces the severity of serialization when many messages contend for the ring.

All of this makes it interesting to try a simple policy that only sends through the photonic ring those messages of small size (control) and through the electrical mesh those messages of large size (data). We will evaluate this simple policy in Section 6.3 under the name of *SIZE*. Notice that the opposite policy (sending data messages on the photonic ring) would be prone to serialization, resulting in high queuing times.

However, this policy has some shortcomings caused by its lack of adaptability to the burst nature of traffic in parallel applications. Under low traffic loads, the fixed-message-size criteria may underutilize the photonic resources (especially in large photonic NoCs), missing opportunities to accelerate and reduce the energy-consumption of other messages. On the other extreme, that is, under high traffic loads or traffic bursts, this policy can completely lose the latency benefit of the photonic NoC when too many short messages contend at a given moment, due to their serialization (especially if there are limited photonic resources).

6.2.2 AVAIL: Ring Availability

This policy sends a message (control or data) through the photonic ring only if the ring is readily available when the transmission is attempted. Hence, the electrical mesh is used only for those messages that find the photonic ring busy. This policy dynamically adjusts the traffic injected in both sub-networks, preventing the shortcomings of the *SIZE* policy.

Since we have to acquire the token before transmitting, we must wait for the token round-trip time (2 processor-cycles in the chosen near-future NoC, see Section 6.3.1) before knowing whether the ring is busy or not. If the token is acquired, then the message is sent through the ring. If it is not, the ring is busy and the message is sent through the mesh (after having incurred an unfruitful wait for the token of 2 processor-cycles).

With this policy, we make sure that messages are never queued for a long time. Rather than waiting for the photonic ring to be free under high traffic load scenarios, a message uses the mesh after one failed attempt of token acquisition. This policy also ensures that every message has a chance of using the photonic ring. Hence, for low traffic loads, we prevent messages from being sent through the mesh if the photonic ring is free, increasing ring utilization and reducing overall energy consumption.

However, even a small number of data message transmissions can now monopolize the photonic ring for long times, causing many more short messages to be sent through the slow mesh. Moreover, when concurrent message transmissions from different nodes are taking place, only one of them is granted access to the photonic ring. The rest are forced to use the mesh, even if it is more efficient to wait a few extra cycles to send their messages through the photonic ring after the first node. To explore all these trade-offs and the detailed effects of this policy, we explore several token-wait delays in the evaluation section under the name *AVAIL*.

6.2.3 Distance Dependent Availability

The energy consumption and latency of a message transmission through an electrical mesh varies depending on the distance between origin and destination. For example, in a 16-core (4×4) CMP, sending a message between opposite corners of the chip requires six intermediate routing operations and six retransmissions through inter-tile electrical links, while transmitting the same message between adjacent tiles requires just one intermediate routing operation and one transmission on the link connecting the tiles. Therefore, communicating distant nodes in a mesh requires much more energy and takes longer. In contrast, the photonic ring is almost insensitive to distance. Thus, the higher the distance, the most energy and latency can be saved if the message is transmitted in the photonic ring instead of in the electronic mesh.

On the other hand, electrical links provide high-bandwidth short-range connectivity efficiently. For instance, a 128-wire electrical link operating at 4 GHz can provide a bandwidth of 64 GB/s in one direction between neighbor tiles in a CMP, which is comparable to a typical photonic waveguide that can manage 64 independent data wavelengths, providing a bandwidth of 80 GB/s when operating at 10 GHz. We would like to exploit this characteristic of electrical meshes with our policies by using the mesh for short-distance transmissions. This way, the photonic ring can be profitably used for long-distance transmissions instead.

To illustrate these ideas with a real scenario, Table 6.3 shows the percentage of messages and link traversals caused by communications between nodes at different distances (in number of network hops) for a uniform random distribution of accesses to a NUCA last-level cache in a 16-core CMP. These values closely match the ones that we observed in the PARSEC benchmarks. Note that transmissions between neighbor nodes (1 hop) account for 20% of messages, but they only generate a relatively small 7.5% of link traversals. In contrast, 5-hop transmissions account for just 6.7% of messages, but since each message traverses 5 links, they generate a significant 12.5% of link traversals. Clearly, 5-hop messages are better

Table 6.3: Message and link traversal distribution in a 4×4 mesh.

Hops	Messages	Link traversals	Aggr. messages	Aggr. link traversals
1	20.0%	7.5%	20.0%	7.5%
2	28.3%	21.3%	48.3%	28.8%
3	26.7%	30.0%	75.0%	58.8%
4	16.7%	25.0%	91.7%	83.8%
5	6.7%	12.5%	98.3%	96.3%
6	1.7%	3.8%	100.0%	100.0%

16-core CMP. 4×4 electrical mesh. Accesses to a NUCA last-level cache. The PARSEC benchmarks follow this pattern with small deviations. Messages that do not leave the tile are not taken into account.

candidates for photonic transmission than 1-hop messages, as they take up less photonic resources (because they are fewer), while providing larger energy and latency savings (because they generate more link traversals in the mesh).

For another illustrative example, take into account the aggregate number of messages and link traversals shown in Table 6.3. Sending 1-hop and 2-hop messages through one sub-network, and the rest through the other, would make each sub-network transmit approximately half of the messages. The photonic ring can transmit any half of the messages with similar latency and energy consumption. This is not the case for the electronic mesh. In the mesh, 1-hop and 2-hop messages would incur less than 30% of the aggregate link traversals of all messages, while the other half of the messages would incur more than 70%. To be precise, 1-hop and 2-hop messages would require traversing 57% fewer links per message than the rest. Therefore, it is a better idea to transmit 1-hop and 2-hop messages through the mesh and the rest through the photonic ring.

Moreover, not all cores are affected by network latency in the same way. Cores in the corners and borders of the chip suffer from longer average network distances and latencies than those in the center [63]. Figure 6.2 shows the average network latency suffered by each core depending on its position in the chip, when communicating with other cores. As a result, a thread running on a corner core executes more slowly than one running on a central tile, due to these network latency differences. This harms parallel application performance, because threads running on *fast* central cores have to wait for those running on *slow* corner cores upon barrier or lock synchronization. In other words, slow cores are the ones to determine the total execution time.

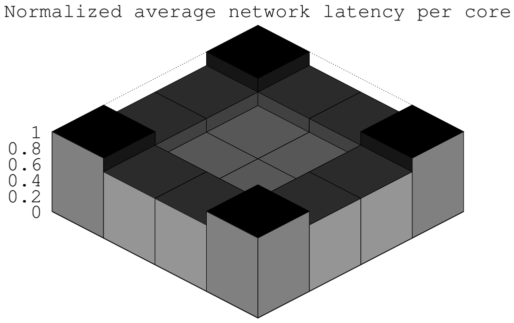


Figure 6.2: Average network latency perceived by cores depending on their position in a 16-core CMP. Cores in the corners suffer 50% longer latencies than cores in the center.

To exploit photonics efficiently, cores far from the center of the chip should make more use of the photonic ring (e.g., only the cores in opposite corners are 6-hops away from each other, and messages between them should go through the photonic ring). This way, the effective network latency differences between cores will shrink, reducing the overhead of synchronization operations on the execution time and boosting performance considerably.

For all these reasons, we have developed a heuristic policy called Distance Dependent Availability (DDA) to decide when to use the photonic ring depending on transmission distance. This policy consists of combining the benefits of using the photonic ring only when it is free, and also using it preferentially for distant endpoint communications. We achieve this mix of goals by allowing a different token-wait time for each particular message that is proportional to the theoretical advantages of using the photonic ring over using the mesh. To calculate the benefits of using the ring, we use the theoretical latency of transmitting each message in the ring (l_p) and in the mesh (l_m) in the absence of other transmissions. Every message is, at first, considered for sending through the photonic ring. If the ring is found idle, the message is sent. Otherwise, the message waits for $(l_m - l_p) \times th$ cycles, where th is a configurable threshold with values between 0 (no wait) and 1 (wait as long as there is any potential benefit in using the ring), before sending the message through the mesh. Small values of th avoid serialization of messages, while large values increase ring utilization. In any case, messages involving distant endpoints wait longer, hence acquiring the token and using the photonic ring more often. In the evaluation section we consider several values for *th* to explore possible trade-offs.

In order to capture the potential benefits of giving differentiated treatment to messages of different sizes, we propose two additional heuristic policies, explained below, that add message size to the variables considered for photonic ring management.

Control DDA (CDDA) consists of using DDA for control messages and AVAIL for data messages. This policy tries to obtain low average message latency in the hybrid network by transmitting many short messages through the ring, prioritizing distant ones, while increasing utilization by also sending data messages when the ring is otherwise idle.

Multi-Threshold DDA (MTDDA) uses DDA for both control and data messages, but different thresholds are used for each message type. A longer threshold is used for control messages to prioritize their transmission in the ring. In this case, we give more importance to ring utilization than in CDDA, as data messages are more likely to be transmitted with MTDDA.

6.2.3.1 Dynamic Thresholds

The burst nature of traffic can make dynamic thresholds useful to avoid unnecessarily long waiting times. Under low traffic loads, a high threshold increases photonic ring utilization without incurring severe message serialization. If the traffic load goes up, many messages will necessarily be sent through the mesh. Then, the threshold can go down to shorten the useless token waiting times of the messages that use the mesh, without reducing the utilization of the photonic ring.

We have explored several dynamic threshold designs, also differentiating by message size, but only marginal improvements were observed in our experiments. Traffic patterns change at a very fine granularity, and adaptive thresholds only provided small benefits in terms of execution time and energy consumption when compared to the simpler policies proposed so far in this chapter. We consider that these results are not significant enough to be discussed in depth. Nevertheless, more elaborated dynamic-threshold mechanisms, able to capture the behavior of the network, could be interesting especially if they could predict future traffic trends and adapt in advance.

6.2.4 Photonic-Electronic Interface

Once photonic and electronic sub-networks coexist, an adequate interface between both technologies is required to apply our management policies. This interface adds little complexity over the ones used in previous works (e.g., Firefly [147]).

In each tile of the CMP, one *pre-photonic buffer* interfaces each traffic source (e.g., L1 or L2 caches) with its associated external input port to the electrical mesh router. Upon injection by a traffic source, those network messages that are candidate for photonic transmission (e.g., control messages in SIZE) are first stored in the corresponding pre-photonic buffer to wait for possible photonic transmission. Eventually, each of these messages is either sent through the photonic ring or transferred to the corresponding external input port to the router for electrical transmission.

At any given moment, only one of the pre-photonic buffers of the node can be *active*, meaning that the message at its head is the one being considered for photonic transmission (i.e., it is the active message). As long as any pre-photonic buffer is active, the token acquisition photodetector of the tile remains on (it is off otherwise). As soon as the token is acquired, the transmission of the active message through the photonic ring starts and the token is reinjected.

In addition, when a message enters a pre-photonic buffer, a countdown timer associated to the entry storing the message is set to the appropriate waiting-time (e.g., AVAIL's 2-processor-cycle token wait). If the timer reaches zero, photonic transmission is ruled out and the message enters the electrical router's external input port. Notice that these timers are not needed by SIZE.

To ensure photonic transmission fairness between a tile's traffic sources, when the active message is transmitted photonically or when its associated counter reaches zero, a round-robin algorithm activates the following pre-photonic buffer containing messages, if any.

On the receiver's end, an ordinary input port to the electronic router is used for receiving the photonic ring output flow to the tile.

6.3 Evaluation

In this section, we discuss the performance of the management policies proposed for photonic-electronic hybrid networks. We have performed two sets of experiments to check out the policies described in Section 6.2. First, we use detailed full-system simulation to test the policies on a photonic ring based on FlexiShare [145] that could be implemented in the near future. Then, we test the

Table 6.4: Features of adapted case study NoCs in which the policies are applied.

	System characteristics		
Solution	**Technology**	**Year**	**Cores**
FlexiShare	35 nm	2014	16
Corona	16 nm	2018	256
Firefly	16 nm	2018	256

	Optical datapath features				
Solution	**N**$_{endp}$	**N**$_{waveg}$	**N**$_{micror.\ res.}$	**Access scheme**	**Phit size**
FlexiShare	16	1	2 K	MWMR	64
Corona	64	256	1024 K	MWSR	256
Firefly	64	256	128 K	SWMR	256

	Electronic features		
Solution	**Concentr.**	**N**$_{links}$	**N**$_{links}^{extra}$
FlexiShare	1	0	24
Corona	4	0	112
Firefly	4	80	32

policies on larger networks based on Corona [173] and Firefly [147] by means of simulations using synthetic traffic. See Section 6.1.2 for a brief description of these networks. Table 6.4 shows the adapted NoCs used in our tests and our estimated target date of availability for these networks. The last column of the table shows the electronic resources (links) that must be added to create the full mesh assumed by our policies. Also, by testing the policies in NoCs of several sizes, we show their general applicability to exploit hybrid networks.

6.3.1 Evaluation Methodology for 16-Endpoint NoCs

For near-future commercial CMPs, we scale down FlexiShare to just one waveguide, resulting in an affordable design that requires just around 2000 microring resonators. Figure 6.1 shows our base hybrid photonic-electrical NoC in a 4×4 CMP. We consider a realistic five ring-cycle full-ring traversal time at 10 GHz for light pulses (i.e., two processor cycles at 4 GHz). All of the data wavelengths of the ring can be simultaneously used by one emitter to communicate with one receiver, and we limit the number of concurrent transmissions in the ring to just one. In all, this FlexiShare-like configuration requires 65 wavelengths for its operation. During destination selection, four wavelengths identify the receiver and one wavelength indicates the size of the message to transmit (one bit is enough to encode the size, as only two sizes exist, corresponding to control and data messages). Sixty-four wavelengths are used for data transmission. One extra wavelength is needed to circulate the single-bit token in which the arbitration mechanism is based (see Section 6.1.3).

As for flow control, we consider that every receiver has enough buffering resources to accommodate the traffic transmitted through a single optical waveguide in the common case. However, buffer overflow may appear eventually, and in that case a NACK signal is sent by the receiver through the data wavelength in the complementary ring portion to the transmission (hence closing the circle without disturbing any other photonic transmission). When the transmitter receives this NACK signal, it backs down, releases the token (if it had not been released yet) and repeats the same transmission procedure again after some time. This flow control mechanism is rarely needed and has little impact in overall performance. More complex mechanisms can be used, but their evaluation is out of the scope of this chapter.

The flexible MWMR configuration of FlexiShare allows for higher utilization of resources and faster single transmissions than MWSR and SWMR rings, especially under unbalanced traffic, as shown by Pan et al. [145] for the SPLASH-2 benchmarks. We have observed that the same holds true for the PARSEC benchmark suite. Even so, this modest short-term MWMR ring is unable to carry all traffic in a 16-core CMP by itself, and we have measured that the execution time for the PARSEC benchmarks is on average 2.2 times higher when using only the ring than when using only the electrical mesh. Thus, our policies should ideally use such a small ring selectively as a sort of *accelerator* when cooperating with the mesh in this scenario.

6.3.1.1 Simulated CMP and Benchmarks Used

The GEM5 simulator [22] was used to perform the tests for 16-endpoint NoCs. The common characteristics of the 16-core simulated CMP can be found in Table 6.5. The L2 cache uses a NUCA design and a directory-based MOESI protocol enforces coherence between the private L1 caches. We have assumed a high-performance 4×4 electrical mesh running at 4 GHz, consisting of bi-directional 1-cycle latency 128-wire links and four-stage pipelined routers. We consider an optimized router architecture in which the destination router requires just one cycle to deliver the message to the appropriate output buffer, instead of four cycles. Under these assumptions, a message transmission between two

Table 6.5: Simulated machine

Processors	16 Alpha cores @ 4 GHz, 2-ways, in-order
L1 Cache	Split I&D. Size: 16 KB, 4-ways, 64 bytes/block Access latency: 1 cycle MOESI coherence protocol (directory cache in L2 cache)
L2 Cache	Size: 1 MB per bank. 16 MB total (NUCA) 8-ways, 64 bytes/block Access latency: 15 cycles
RAM	4 GB DDR2 DRAM 16 3D-stacked memory controllers
Interconnection - Mesh	4 GHz, 2D mesh: 4×4 16 byte links. Latency: 1 processor-cycle/link 4-processor-cycle pipelined routers Flit Size: 16 bytes Control/Data packet size: 8/72 bytes (1/5 flits) Dynamic energy (1-hop switch+link): 282 pJ/flit Static power (switch+link): 52.7 mW
Interconnection - Photonic	10 GHz MWMR Photonic Ring. 3D-stacked 65 wavelengths Latency: 2 processor-cycle round-trip time 2 processor-cycle minimum transmission time on idle ring (no token wait, closest node) 6 processor-cycle maximum transmission time on idle ring (roun-trip time token wait, furthest node) Flit Size: 8 bytes. Control/Data packet size: 8/72 bytes (1/9 flits) Dynamic energy: 0.41 pJ/bit Static power (laser+microrings): 318 mW

adjacent routers requires just 6 cycles for the first flit to go from the initial buffer in the mesh to the final buffer (one 4-cycle router traversal, one 1-cycle link traversal, and a 1-cycle router delivery), while between the most distant routers it requires 31 cycles (six 4-cycle router traversals, six 1-cycle link traversals, and one 1-cycle router delivery).

On its part, transmitting on the 10 GHz MWMR photonic ring when it is idle requires to acquire the free token (up to five ring cycles), activate the destination's receivers (three ring cycles) and then reach the destination with the first photonic pulse (up to five ring cycles). In the case of control messages, this is enough to transmit the 8-byte message. In the case of data messages, a second photonic pulse carries the eight-byte requested word (the first pulse contains an 8-byte header). The rest of the data block is transmitted in consecutive photonic pulses.

In the most favorable case (no wait for the token and a transmission to the closest node), an idle photonic ring provides a 2-processor-cycle transmission latency (rounded up) for control messages or for the requested word of data messages. In the most unfavorable case (round-trip time wait for the token and transmission to the farthest node) this latency increases to 6 processor-cycles. These latencies are more conservatives than others reported in the literature [145]. In comparison, the idle 4×4 electrical mesh requires up to 31 processor-cycles between the most distant destinations, as calculated previously.

The timing and power parameters of the electronic NoC are derived from Orion 2.0 [94] using a 32 nm silicon process. We consider state-of-the-art optical devices [188] and their behavior in our reference architecture.

We have used the PARSEC benchmark suite with the medium-sized working sets to perform this study [21]. Table 6.6 shows the photonic policies evaluated. We evaluate the SIZE policy, the AVAIL policy with three different waiting times (2, 6 and 10 processor cycles), the DDA and CDDA policies with three different thresholds (25%, 50% and 75%) and the MTDDA policy with two different configurations (60%–40% and 75%–25% thresholds for control and data messages). Further details on these policies can be found in Table 6.6 and in Section 6.2.

6.3.2 Evaluation Methodology for 64-Endpoint NoCs

We also evaluated our policies on larger NoCs. For this, Corona [173] and Firefly [147] were chosen as good examples of future NoCs for 256 cores (4 cores per endpoint). The NoC parameters were set to match those of the original works, unless stated otherwise. Table 6.7 describes the five synthetic traffic patterns used

Table 6.6: Evaluated policies

Configuration	**Messages on photonic ring**	**Messages on mesh**
mesh	none	all
SIZE	control messages (8 bytes)	data messages (72 bytes)
AVAIL-2	token acquired within 2 cycles	other messages (2-cycle delay)
AVAIL-6	token acquired within 6 cycles	other messages (6-cycle delay)
AVAIL-10	token acquired within 10 cycles	other messages (10-cycle delay)
DDA-25	token acquired within $(l_m - l_p) \times 0.25$ cycles	other messages $((l_m - l_p) \times 0.25$ delay)
DDA-50	token acquired within $(l_m - l_p) \times 0.50$ cycles	other messages $((l_m - l_p) \times 0.50$ delay)
DDA-75	token acquired within $(l_m - l_p) \times 0.75$ cycles	other messages $((l_m - l_p) \times 0.75$ delay)
CDDA-25	control if token acquired within $(l_m - l_p) \times 0.25$ cycles data if token acquired within 2 cycles	other messages $((l_m - l_p) \times 0.25$-cycle delay for control, 2-cycle delay for data)
CDDA-50	control if token acquired within $(l_m - l_p) \times 0.50$ cycles data if token acquired within 2 cycles	other messages $((l_m - l_p) \times 0.50$-cycle delay for control, 2-cycle delay for data)
CDDA-75	control if token acquired within $(l_m - l_p) \times 0.75$ cycles data if token acquired within 2 cycles	other messages $((l_m - l_p) \times 0.75$-cycle delay for control, 2-cycle delay for data)
MTDDA-60-40	control if token acquired within $(l_m - l_p) \times 0.60$ cycles data if token acquired within $(l_m - l_p) \times 0.40$ cycles	other messages $((l_m - l_p) \times 0.60$-cycle delay for control, $(l_m - l_p) \times 0.40$-cycle delay for data)
MTDDA-75-25	control if token acquired within $(l_m - l_p) \times 0.75$ cycles data if token acquired within $(l_m - l_p) \times 0.25$ cycles	other messages $((l_m - l_p) \times 0.75$-cycle delay for control, $(l_m - l_p) \times 0.25$-cycle delay for data)

l_m = idle mesh latency, l_p = idle photonic ring latency. All latencies in processor cycles @ 4 GHz

Table 6.7: Synthetic traffic patterns.

Traffic name	Details
Uniform	Uniform random traffic
Transpose	$(i,j) \Rightarrow (j,i)$
Bitcomp	dest id = bit-wise-not(src id)
Neighbor	Randomly send to one of the source's neighbors
Tornado	$(i,j) \Rightarrow ((i+\lfloor X/2 \rfloor - 1) \mod X, (j+\lfloor Y/2 \rfloor - 1) \mod Y)$

in the tests, which are inspired by the work of Fallin et al. [50]. Of these, uniform traffic is the most similar to the one observed in real applications using a NUCA cache.

The cumulative injection rates for the four concentrated processors, in packets per cycle, are used as the load metric of our tests. Short (64-bit) and long (576-bit) packets were injected randomly. Four-cycle routers were used in the mesh.

6.3.3 Results for 16-Endpoint Hybrid NoCs

6.3.3.1 Execution time analysis

Figure 6.3 shows the execution times of all the configurations tested, using the PARSEC benchmark suite. Also, the relative standard deviation of the network latency suffered by the cores is shown ("network latency relative standard deviation (%)"). A high value means that some cores suffer higher average network latencies than others. The average data is shown on the lower right corner.

In general, all the photonic management policies reduce execution time compared to the baseline electrical mesh. Also, the trends shown by each policy remain stable across all benchmarks.

The SIZE policy (see Section 6.2.1), which transmits every control message through the ring, reduces execution time by 27% on average with respect to the baseline mesh, thanks to a reduction in the latency of control messages.

The AVAIL policy (see Section 6.2.2), which only sends messages when the token can be acquired within a fixed number of cycles, shows different behavior than SIZE. To start with, the execution time is noticeably higher than SIZE's, as AVAIL-2 reduces execution time by just 21% compared to the baseline. The reason is the larger average size of the messages transmitted through the ring (control and data in AVAIL, only control in SIZE) that causes fewer messages to be accelerated compared to SIZE. As the number of waiting cycles increases

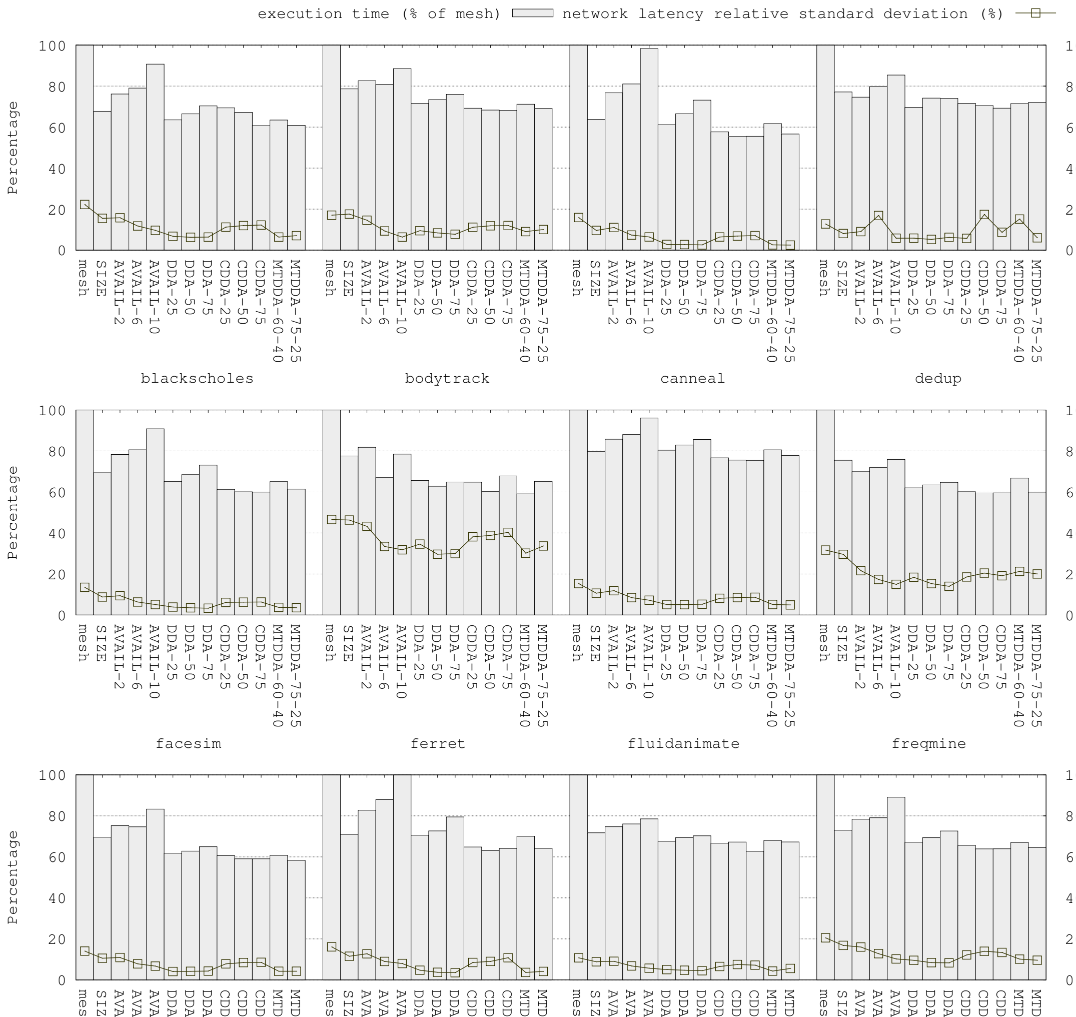


Figure 6.3: Execution time. Normalized to electronic mesh.

(AVAIL-6 and AVAIL-10), the average network latency increases too, resulting in average reductions of just 20% and 11% in execution time, regarding the baseline.

Distance-based policies (see Section 6.2.3) make more efficient use of photonics and improve the performance results of SIZE and AVAIL. DDA-25 reduces execution time by 33% with respect to the baseline. As the policy threshold goes up, the execution time reduction goes down to just 30% (DDA-50) and 27% (DDA-75). This performance drop is caused by an increase in the average waiting times to transmit when using higher thresholds.

CDDA, which uses DDA for control messages and AVAIL-2 for data messages, improves performance with respect to pure DDA by reducing the latency of a higher amount of short messages. CDDA provides the highest reduction in execution time of all policies. CDDA-25 reduces execution time by 33%, and this value increases to 36% for CDDA-50 and CDDA-75.

Finally, MTDDA, which uses different thresholds for control and data messages, performs close to CDDA, with 33% and 35% lower execution times than the baseline for MTDDA-60-40 and MTDDA-75-25, respectively. These policies allow some waiting for data messages, based on distance, in order to achieve a balance between execution time speedup and photonic ring utilization.

Figure 6.3 also shows that our policies reduce the standard deviation of the network latency suffered by the cores. This reduction is especially noticeable for DDA and MTDDA, where it reaches 60%. In specific benchmarks, such as canneal, this reduction goes up to 85%, which means that all the cores perceive much more homogeneous network latencies. This has the important benefit of reducing the waiting times for thread synchronization, which helps improve performance.

6.3.3.2 Network Latency Analysis

Figure 6.4 shows the average latencies for message transmissions in the electrical mesh ("av. mesh latency") and in the photonic ring ("av. ring latency"). Also, the overall message transmission latency of the chip is shown ("av. net. latency"). Notice that, although the frequencies of these networks are different (4 GHz for the electrical mesh and 10 GHz for the photonic ring), all the data in the graphs are plotted at processor frequency (4 GHz) to give a comparable view of both networks. We also show the theoretical latency in the (idle) mesh for the messages that were finally sent through the photonic ring instead ("av. latency avoided by ph. ring"). This value reveals how much gain was obtained, in number of

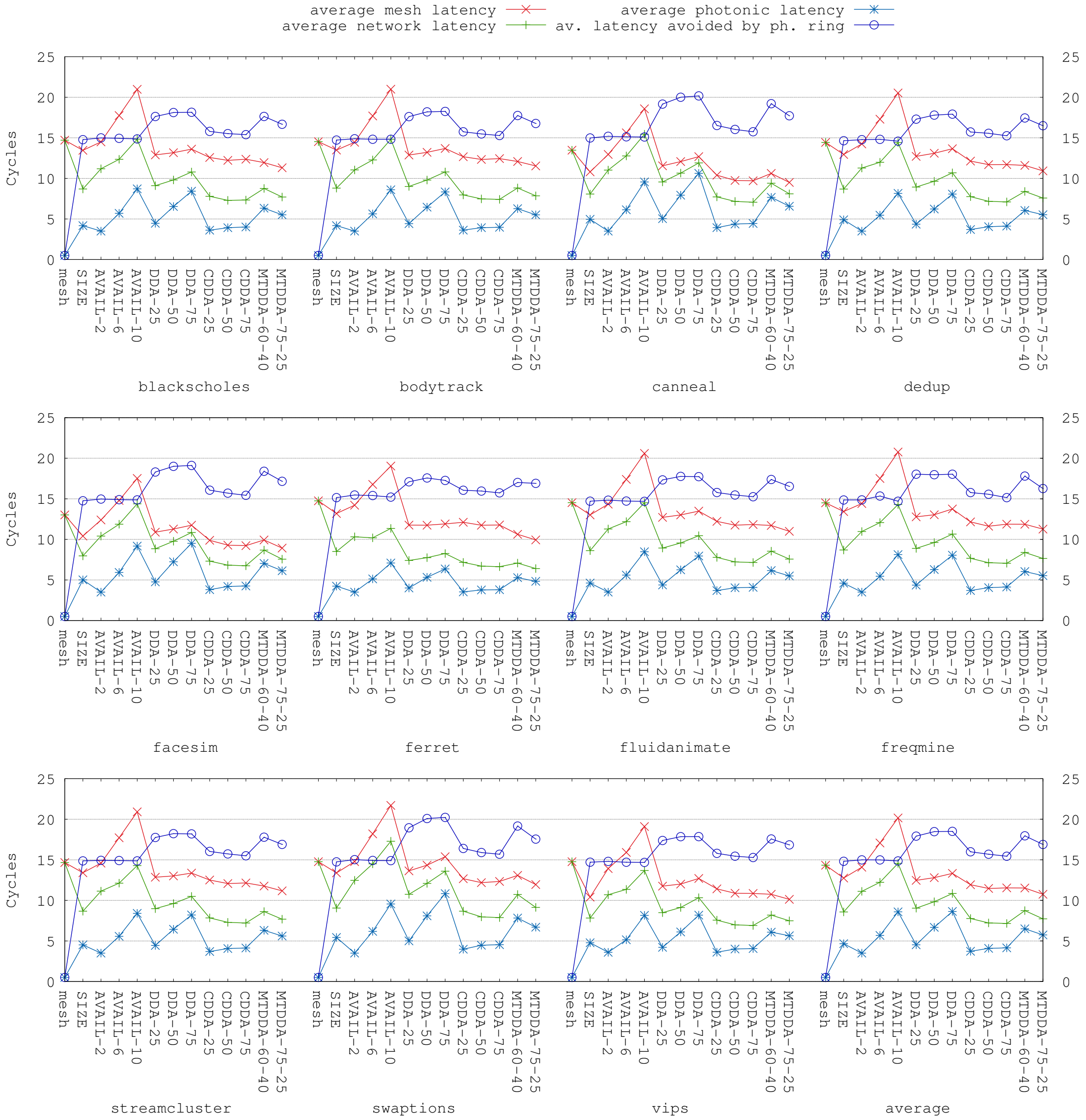


Figure 6.4: Network latency. Normalized to electronic mesh.

electrical retransmissions avoided, each time that the photonic resources were used instead of the mesh.

In SIZE, all control messages are sent through the photonic ring, reducing the network latency by 40% on average. In general, the waiting time for acquiring the ring is low (an average of 1.2 processor cycles) thanks to the small size of the messages transmitted, which avoids long serializations.

In AVAIL, the sending of data reduces the number of messages sent through the photonic ring compared to SIZE. Therefore, the latency reduction provided by the photonic ring is applied to fewer messages, resulting in a lower 23% average network latency reduction, which explains why AVAIL cannot reduce execution time as much as SIZE does. As the waiting cycles are increased (AVAIL-6 and AVAIL-10), we observe the expected increases in the average latency of the ring, in the average latency of the mesh (because messages need to wait longer for the ring before using the mesh), and in the overall network latency. For instance, the average network latency of AVAIL-10 is the same as the baseline. Nevertheless, AVAIL-10 reduces the network-latency-per-core standard deviation by 50% on average, compared to the baseline (Figure 6.3), speeding up thread synchronization as a result. While the absolute core busy time remains similar in both the baseline and AVAIL-10, the idle time gets noticeably reduced in AVAIL-10, making execution times go down by 11% on average.

DDA provides 37%, 31% and 24% lower average network latencies than the baseline for 25%, 50% and 75% thresholds, respectively, because the photonic ring is preferentially used to send long-distance messages thanks to the distance dependent wait for the token. The mesh latency avoided by the photonic ring is higher than in the previous policies, resulting in a lower average latency for the mesh, which is now mainly used for short-distance messages. As the thresholds rise, the waiting times for the ring go up, explaining the increasing latencies.

In CDDA, many short control messages are sent through the photonic ring. The distance-dependent thresholds manage to keep high the mesh latency avoided by the ring. The avoided mesh latency is lower than DDA's because now most control messages use the photonic ring, which includes many messages between close endpoints. Overall, the latency reduction is the highest of any policy, with 46%, 49% and 50% latency reductions for thresholds of 25%, 50% and 75% respectively, thanks to the higher number of messages benefited by a high latency reduction.

MTDDA shows an intermediate profile between DDA and CDDA thanks to the two different thresholds for control and data. MTDDA-75-25 provides the highest latency reduction of the two MTDDA policies: 46%.

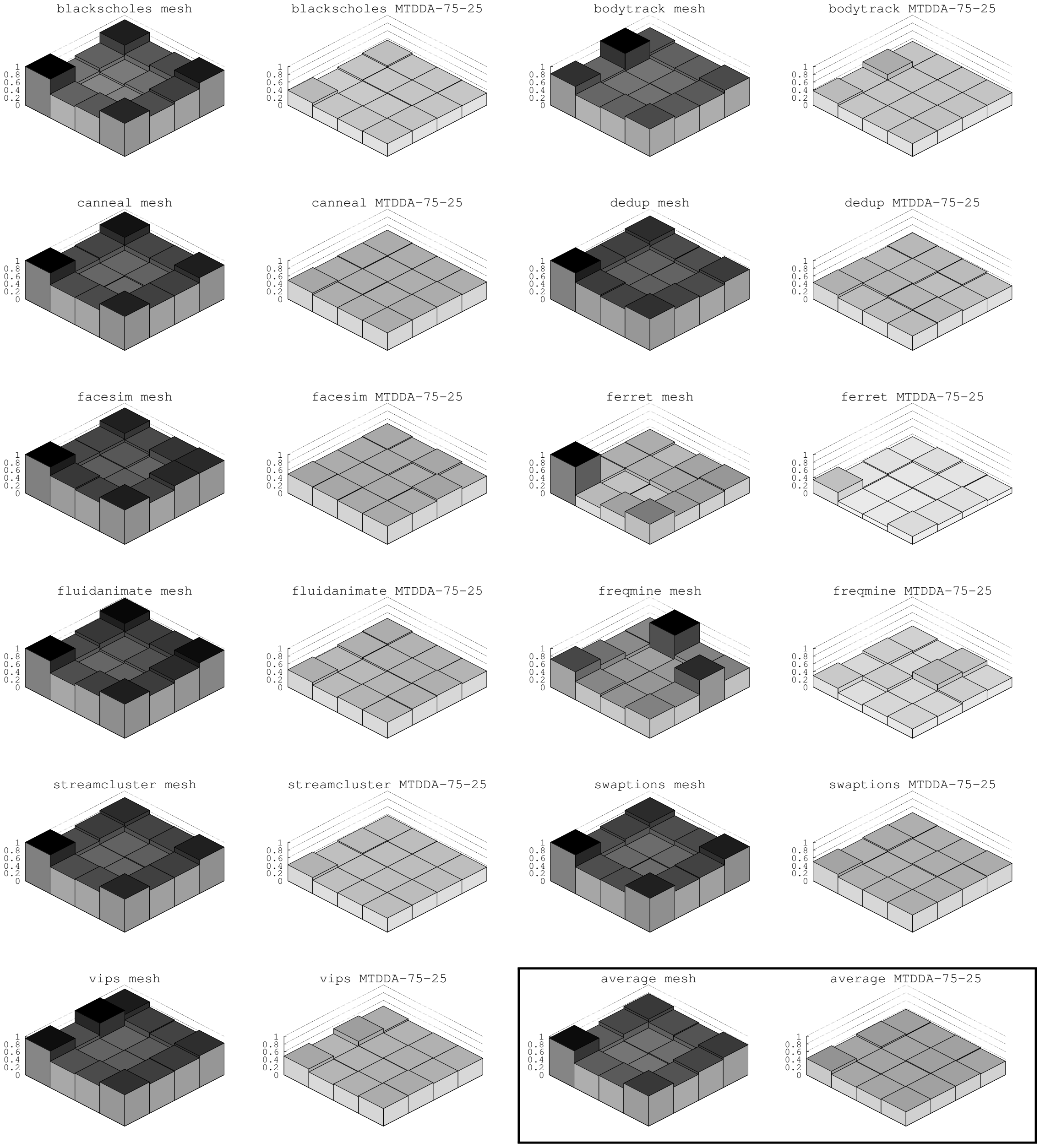


Figure 6.5: Total network latency suffered by each core in the critical path of L1 cache misses. Results for electrical mesh and MTDDA-75-25 for PARSEC benchmarks and average of all benchmarks. The data is normalized to the core with the highest network latency in the mesh.

Figure 6.5 shows the overall absolute latency suffered by each core for the electrical mesh and for MTDDA-75-25 in each benchmark. The pattern shown by the mesh in most benchmarks matches the one previously shown in Figure 6.2 for NUCA LLC accesses. Figure 6.5 gives a clear view of how those cores that suffer longer latencies in the mesh are the most benefited by MTDDA-75-25 (corners and borders of the chip). The threads running on these cores get a higher speed-up and their performance can now match that of the threads running in the central cores, preventing them from slowing down the execution of parallel applications. This causes a noticeable portion of the acceleration of applications seen in Figure 6.3.

6.3.3.3 Network Energy Consumption Analysis

Figure 6.6 shows the energy consumption of the network, split in electrical mesh and photonic ring. Also, the photonic network usage is shown. We have included in this graph the percentage of messages that were transmitted through the photonic ring ("% of msgs. in ph. ring"). A higher percentage of messages does not necessarily imply higher utilization of the ring, as utilization also depends on how many messages of each size (control and data) were transmitted. This is further detailed in Figure 6.6 by showing the percentage of messages that are at once control messages and transmitted photonically as "% of msgs. in ph. ring (control)". The difference between this value and "% of msgs. in ph. ring" accounts for data messages transmitted photonically.

In SIZE, the ring is empty in those cycles in which there are no control messages ready to be transmitted, as well as in those cycles that the token has to travel until it reaches a far node that is ready to transmit a control message, spared by those intermediate nodes that only have data messages. Even though more than 50% of messages are transmitted photonically, they are all small, causing an underutilization of the photonic ring (18% usage). Nevertheless, SIZE reduces the energy consumption of the network by 17% with respect to the baseline.

In AVAIL, the sending of data provides more opportunities to make use of the photonic ring. The ring usage rises to 48% in AVAIL-2, and the energy reduction reaches 28%. Each data message sent with AVAIL uses the ring for the equivalent time of nine control messages, explaining the higher ring usage even though less messages use the ring than in SIZE. In addition, increasing the waiting cycles also causes an increase in the usage of the ring (65% and 69% for AVAIL-6 and AVAIL-10) which results in a noticeable reduction of the energy consumption

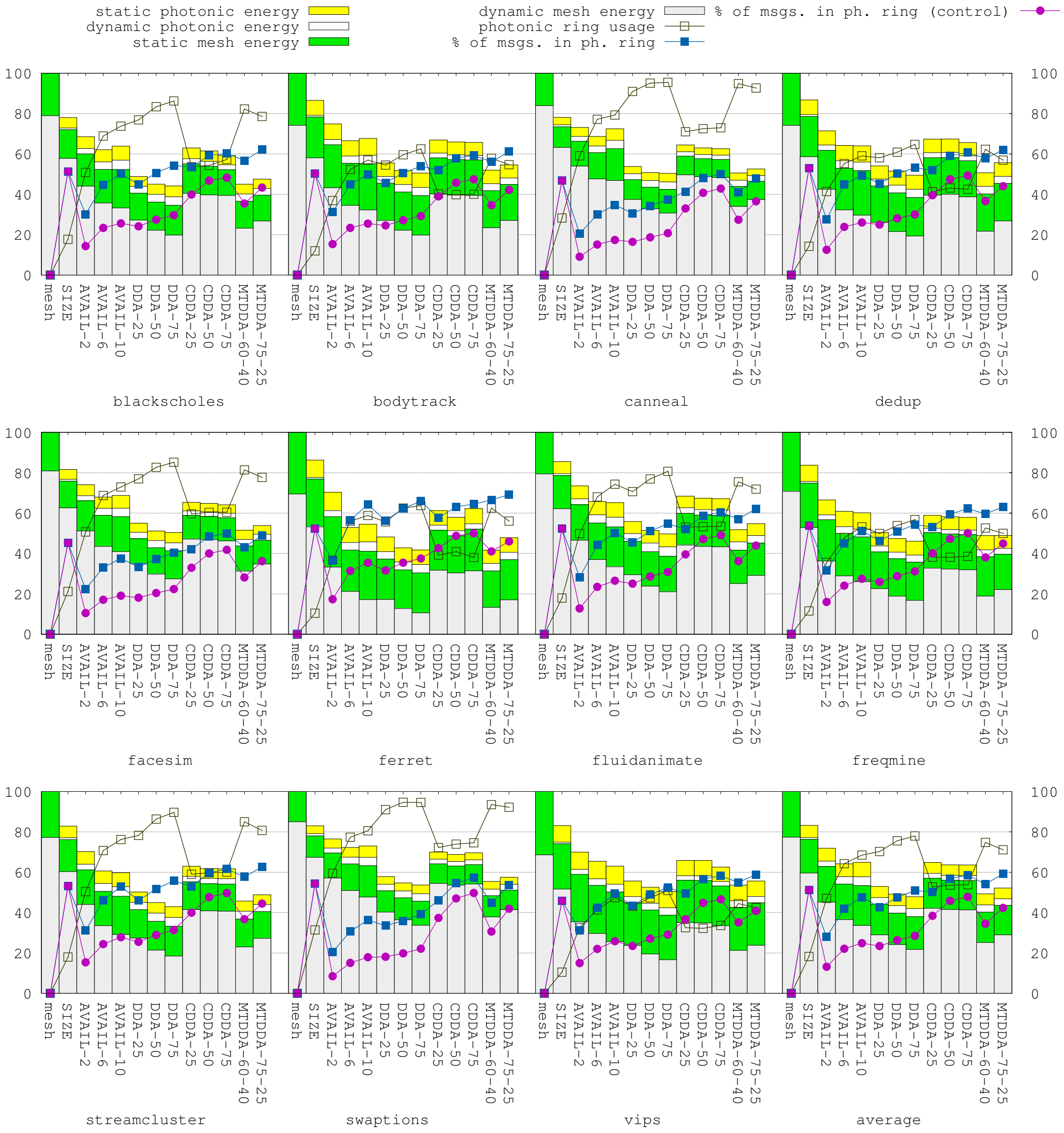


Figure 6.6: Network energy consumption (normalized to electronic mesh) and photonic ring usage.

of the network (36% for both). We can conclude that AVAIL prioritizes energy reduction over execution time when compared to SIZE. Also, the amount of waiting cycles provides a way to tune between lower execution time and lower network energy consumption.

DDA achieves the highest reductions in network energy (47%, 50% and 52% for 25%, 50% and 75% thresholds), since its photonic ring usage is the highest and the ring is used for messages between distant endpoints. The remaining messages require fewer retransmissions in the mesh, saving the most energy overall. Similarly to AVAIL, DDA's thresholds provide a trade-off between execution time and network energy consumption.

In CDDA, which provides the fastest execution time by sending control messages through the photonic ring with high probability, a reasonably high utilization of the ring is achieved by also sending data messages when the ring is found available (see the percentages of messages in Figure 6.6). CDDA's ring usage is 53%, 54% and 55% for 25%, 50% and 75% thresholds, which is lower than DDA's (as most messages transmitted photonically are short in CDDA) but much higher than SIZE's. The network energy consumption reductions are 35%, 36% and 36%, which are smaller than DDA's.

MTDDA results in a good trade-off between DDA and CDDA. By using different thresholds for control and data we can still prioritize the sending of short messages in order to reduce execution time, like CDDA, while retaining the ability to achieve a high utilization of the photonic ring with data messages (76% and 72% for MTDDA-60-40 and MTDDA-75-25), like DDA. The network energy reductions are 51% and 48% for MTDDA-60-40 and MTDDA-75-25. We believe that MTDDA is the most versatile policy, providing good results in both execution time and energy consumption. In terms of execution time, MTDDA almost matches CDDA, and in terms of network energy consumption, MTDDA is closer to DDA than to CDDA.

6.3.4 Results for 64-Endpoint Hybrid NoCs

In the case of larger hybrid networks comprising 64 endpoints, based on Firefly and Corona, the chosen policy to manage them was MTDDA-75-25 due to its good results in the 16-endpoint scenario just analyzed. We use the names *Firefly** and *Corona** to refer to the hybrid networks managed with MTDDA-75-25, in contrast to *Corona* and *Firefly*, which we use to refer to the original versions of these NoCs. We also simulated one of the simpler policies, AVAIL-6, as a baseline to compare against MTDDA-75-25 in order to estimate which benefits come from

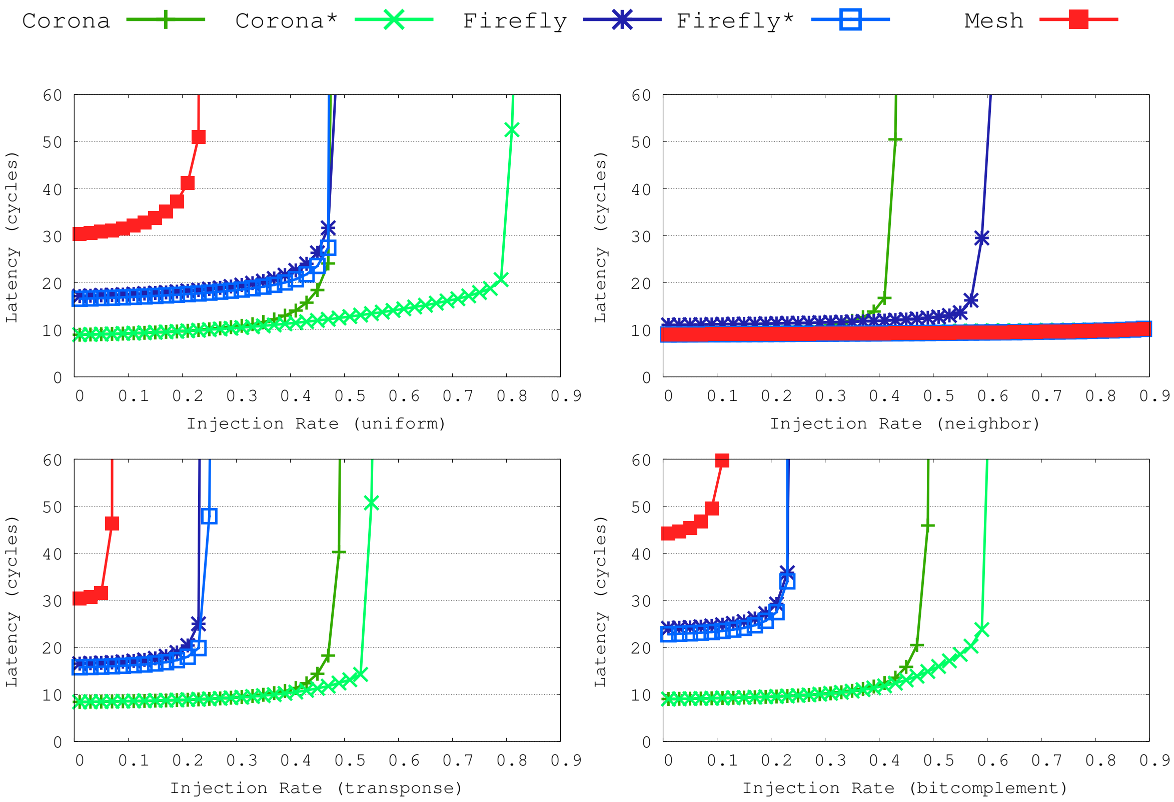


Figure 6.7: Load latency curves for uniform, neighbor, transpose and bitcomp traffic. 256-bit photonic datapath width.

the extra network resources (added to create the hybrid NoC) and which from the use of a smarter policy such as MTDDA-75-25. We do not show AVAIL-6 in the graphs for clarity, but we refer to the hybrid networks managed with AVAIL-6 by the names *CoronaAV* and *FireflyAV* in our analysis.

Figure 6.7 shows the results for four synthetic traffic patterns. Notice that, under uniform traffic, the throughput of Corona* (0.80 msgs/cycle) is 18% larger than the sum of those of Corona (0.46 msgs/cycle) and the mesh (0.22 msgs/cycle) separately. MTDDA-75-25 exploits the best features of the mesh for short-distance communication. By selectively using the mesh for messages to close destinations (which require few retransmissions and are fast), and avoiding messages to distant destinations (which require many retransmissions and are slow), the mesh is able to transmit many more messages, and faster, with MTDDA-75-25 than when working alone. For long distances, MTDDA-75-25 uses photonics, which is fast and has similar throughput regardless of the distance of transmissions. This increases the overall throughput of the hybrid network and at the same time keeps a very low latency for Corona*. CoronaAV is not so efficient, as it takes neither distance nor message size into account to modulate the use of the sub-networks, resulting in a throughput (0.58 msgs/cycle) that is higher than that of Corona but lower than sum of those of Corona and the mesh. For the same reasons, CoronaAV does not have the latency benefits of Corona*, as AVAIL-6 does not prevent slow long-distance transmissions in the mesh. As a result, CoronaAV is generally closer to the mesh than to Corona in terms of message latency.

Neighbor traffic has the lowest latency and highest throughput possible for any traffic pattern in a mesh (>1.00 msgs/cycle). Corona (0.42 msgs/cycle) and Firefly (0.60 msgs/cycle) yield poor throughputs (Firefly has an advantage against Corona because of the intra-cluster electronic links) compared to Corona* and Firefly* (>1.00 msgs/cycle) which benefit from the electronic links to neighbors. The 32 extra links of Firefly* make a big difference over Firefly. CoronaAV and FireflyAV also have high throughputs, but their latency is 50–60% higher than Corona* and Firefly* due to the unnecessary photonic waiting times introduced by AVAIL-6.

In transpose and bitcomp traffics, each origin has a predefined destination for its messages, at a fixed distance, giving less flexibility for MTDDA-75-25 to arbitrate. Even so, Corona* (0.54, 0.60 msgs/cycle) has higher throughputs than CoronaAV (0.50, 0.51 msgs/cycle) and Corona (0.48, 0.48 msgs/cycle), because MTDDA-75-25 uses the mesh efficiently for those origin-destination pairs with shorter distances and, in any case, because it takes into account the dynamic photonic-link occupation status to decide the most-effective route for each message.

Firefly* and FireflyAV do not benefit from the extra 32 links with respect to Firefly in uniform, transpose, bitcomp nor tornado traffics, confirming the observations by Firefly's authors [147]. The physical design of Firefly forces electronic intra-cluster transmission for every message (except when a direct photonic path exists between origin and destination, which is the case for just 7 out of every 63 origin-destination pairs), flooding the intra-cluster electronic links regardless of the policy managing the sub-networks. In fact, this intra-cluster bottleneck limits the achievable degree of utilization of the inter-cluster photonic resources, which never rises over 37% in transpose traffic.

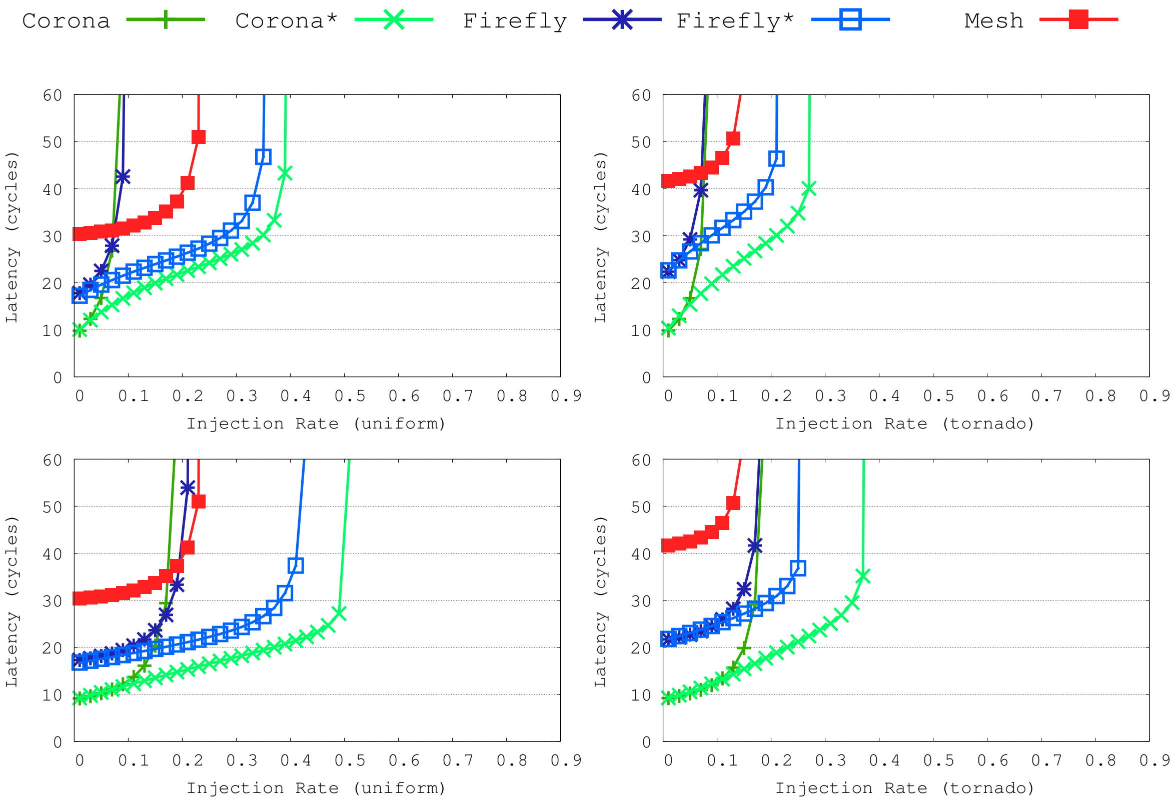


Figure 6.8: Load latency curves for uniform and tornado traffic, using 32-bit (top) and 64-bit (bottom) photonic datapath width.

6.3.4.1 Effects of Reduced Datapath Width

For completeness in our analysis, we have also simulated versions of the Corona and Firefly networks with a photonic datapath width reduced from 256 bits to 64 and 32 bits. These configurations are intended to fill the gap in our analysis between the short-term ring evaluated in Section 6.3.1 and the long term proposals of Firefly and Corona just evaluated. Figure 6.8 shows the results for uniform and tornado traffic for 32-bit and 64-bit datapaths. Note that the electric mesh shows superior or similar throughput to Corona and Firefly for these photonic datapath sizes.

In uniform and tornado traffics with a 32-bit datapath, Corona* (0.39, 0.28 msgs/cycle) has 30% and 33% higher throughput than the sum of Corona (0.08, 0.07 msgs/cycle) and the mesh (0.22, 0.14 msgs/cycle). More importantly, MTDDA-75-25 uses photonics smartly (for long distances), managing to keep the

Table 6.8: Percentage of packets optically transmitted per distance (network hops). Uniform traffic, 0.15 msgs/cycle injection rate.

NoC	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Firefly	29	59	81	94	100	100	100	100	100	100	100	100	100	100
Firefly*	0	13	28	49	64	70	75	79	82	84	85	85	86	88

latency of the hybrid network much lower than that of the mesh, even though the mesh is providing most of the throughput of Corona* (for short distances, with low latency). AVAIL-6 fails to do any of this, resulting in CoronaAV suffering average latencies higher than the mesh and yielding noticeably lower throughputs (0.25, 0.17 msgs/cycle) than Corona*. These results confirm that MTDDA-75-25 increases the throughput of the hybrid network noticeably, over the sum of the parts, and achieves low latencies with any combination of photonic and electronic resources. The performance of AVAIL-6 shows that less elaborated policies cannot do any of this. Another representative result is that, under uniform traffic, Corona* with a 64-bit datapath (0.49 msgs/cycle) has slightly higher throughput than Corona with a 256-bit datapath (0.48 msgs/cycle), while CoronaAV with the same 64-bit datapath performs far worse (0.35 msgs/cycle).

Note that even though Firefly contains most of the mesh, its throughput with a 32-bit photonic datapath is noticeably worse than that of the mesh. The design of Firefly forces the use of photonics for most messages, even between neighbors, making photonics the limiting factor and misusing the intra-cluster electronic links. This is the opposite case to the one noted with a larger 256-bit photonic datapath, in which the flooded intra-cluster links were the bottleneck limiting the efficient use of photonics. Table 6.8 shows the percentage of messages that used photonics in uniform traffic at an injection rate of 0.15 msgs/cycle. The absence of inter-cluster electronic links forces Firefly to use photonics for 29% of the transmissions to neighbors and for most transmissions at a distance of 2 network hops, creating the bottleneck. In contrast, Firefly* never needs to use photonics for transmissions to neighbors (it uses the mesh links instead) and its usage of photonics grows gracefully with the distance. With uniform traffic and a 32-bit datapath, Firefly*'s throughput (0.33 msgs/cycle) is four times larger than Firefly's (0.08 msgs/cycle), by just adding the 32 extra inter-cluster links.

6.4 Related Work

Since silicon-photonic integration became a feasible solution for CMP interconnects, a myriad of photonic networks have been proposed as a solution to the lack of scalability of electrical NoCs. Many nanophotonic-based network topologies have been studied, from simple photonic rings [145, 173, 178] that operate like a crossbar, to complex articulated topologies [148, 160] that require or combine different transmission technologies, to logical all-to-all interconnect designs [138]. Instead of proposing new topologies, our work is the first to provide generally applicable fine-grain policies to exploit the best features of nanophotonics and electronics working together.

Some complex photonic interconnects use supporting circuit-establishing electrical networks [148, 160]. This limits severely the latency and energy advantages of nanophotonics in scenarios like hardware-cache-coherent CMPs with memory-block-grain network communication. We focus on a simple photonic structure (ring) instead, and show that, if properly managed, it can potentially deliver large latency and energy improvements without needing big investments in complex and/or articulated photonic structures.

In CMP designs integrating photonic resources, the assumed NoC is often hybrid to some degree. For instance, concentration is present in many works [109, 145, 173, 178]. That means that a number processing elements share each photonic endpoint, using electronic communication between them. These interconnects use both electric and photonic technologies, whose interaction is decided at design time. Like them, we focus on realistic baselines that require photonics and electronics to optimize performance, but unlike them, we focus on a more efficient dynamic management of the electric/photonic NoC resources at a message granularity.

Also, different arbitration and QoS policies [146, 172] have been proposed, as well as static traffic selection policies [15], to make the most of the limited resources of the shared photonic medium. On our part, we have shown how dynamic management policies based on distance between endpoints enable more fairness in the chip by reducing the network latency differences suffered by tiles in different positions of the chip.

Finally, the use of long-range on-chip wireless links has been proposed to increase energy efficiency [123], with an approach similar to but less flexible than our policies. In addition, our policies could be adaptable to such transmission technology.

6.5 Conclusions

In this chapter, we have shown the importance of using adequate management policies to enable efficient use of any amount of photonic resources in a hybrid photonic-electronic network. We have proposed the first fine-grain dynamic policies to enable such management. We have tested these policies both on an affordable photonic ring for near-future CMPs and on large ring-based photonic networks, obtaining large performance improvements and energy consumption reductions.

The proposed message-granularity policies are based on distance between endpoints, ring availability and message size. By using photonics for the messages most likely to benefit from it (distant, short, and keeping low waiting times), we reduce the number of electric mesh retransmissions (that cause large energy consumption and latency). At the same time, we prevent severe message serialization on the photonic ring due to queuing by resorting to the mesh when necessary, and preferably for short-distance messages. In addition, these policies level out the network latencies suffered by all the cores in the chip compared to an electrical mesh, preventing the existence of *slow* cores that increase execution times. This results in an additional performance boost thanks to quicker thread synchronization that reduces processor idle times.

Among the proposed policies, a performance oriented policy (CDDA-75) reduces execution time by 36%, and an energy oriented policy (DDA-75) reduces network energy consumption by 52% for the PARSEC benchmark suite in a 16-core CMP. In addition, we propose a balanced policy (MTDDA-75-25) which reduces execution time by 35% and network energy consumption by 48%. Larger 64-endpoint NoCs also show far superior throughput and lower latency if managed by an appropriate policy such as MTDDA-75-25, with a throughput that is higher than the sum of the throughputs of the sub-networks when working alone, and a latency that is very low as a result of transmitting each message through the appropriate sub-network.

Chapter 7

Conclusion and Future Ways

For some years now, the evolution of computing has been limited by the increasing difficulty to improve processor performance while keeping a reasonable power budget. This limitation arose because, to be able to exploit high levels of ILP, single-core processor designs had to be increasingly complex. This complexity had as a consequence an increase in power consumption, which made the power costs of maintaining such design trend become unreasonable. Also, the end of the energy benefits of classic transistor scaling made technology advancements unable to reduce the energy-per-operation of processors at the same rate as in the past. The coincidence in time of both circumstances created a large barrier to computing evolution, widely known as the *power wall*.

In order to alleviate the situation, multi-core designs that exploit TLP have been widely adopted as an energy-efficient alternative to the dynamic extraction of ILP to improve performance. The reason justifying this shift in design is that computing power can potentially grow linearly with area and power consumption by just replicating (almost) identical cores in multi-core designs such as tiled-CMPs. In contrast, increasing the complexity of single-core processors to exploit more ILP presents much worse energy scalability.

Unfortunately, there are also large obstacles on the way to realizing such potential benefits of multi-cores, preventing their energy scalability from matching our expectations in practice.

First, maintaining hardware cache coherence, necessary to enable the convenient shared memory parallel programming model with good performance in the presence of private caches, becomes very expensive for large core counts. The area requirements of directory-based cache coherence, which is regarded as the most scalable among realistic schemes nowadays, scale quadratically with the number of cores, breaking the multi-core ideal of linear area growth.

Second, the design and operation of current electrical networks on chip (NoCs) increase the average number of retransmissions required by each message as the number of cores grows. This in turn makes network traffic increase faster than the number of cores, rendering current NoC designs non scalable in terms of energy.

Breaking down these barriers will require extensive research efforts by computer architects. In this thesis, we have focused our efforts on taking steps towards solving some of the key issues of the scalability of multi-cores, resulting in contributions that should improve the scalability of cache coherence and NoCs. The following are the most remarkable conclusions drawn from our work:

- **We have shown that it is possible to reduce the size of the sharing information and increase data proximity without increasing the pressure on the LLC (Chapter 3).** To do so, we have proposed and evaluated a novel coherence scheme with the chip statically divided in areas in which deduplicated data is stored only once in the shared level of cache and data proximity is increased thanks to the use of *providers* close to the requestors. This scheme results in reduced area footprint for cache coherence. Based on it, two protocols have been proposed that use prediction to send memory requests straight to an adequate provider. We have shown that our protocols achieve a reduction of 59–64% in the area required to store directory information for a 64-tile CMP with just 4 areas, which reduces static power consumption by 45–54% and improves scalability. They also reduce dynamic power consumption by up to 38% for the most representative workload for the scenario considered (apache in a consolidated server). In addition, our proposals noticeably outperform an optimized directory protocol in most workloads thanks to the use of providers, with speedups up to 6%. Moreover, as the number of cores increases in CMPs, the advantages of these schemes become even more noticeable, with both the area taken up by coherence information and the distance required to reach a provider going down. As examples, in a 256 core CMP with 64 areas, the average distance traveled by messages goes down from 32 links with an ordinary directory protocol to just 2.4 with our proposals, and with 1024 cores and 16 areas, the storage overhead of directory information is reduced by 90.5%.

- **We have proven that it is possible to notably reduce the average number of network links traversed to access partially-shared caches by making each core access the LLC banks surrounding it (Chapter 4).** We call our proposal to do this DAPSCO. DAPSCO is applicable in conjunction with any network topology. The cost of DAPSCO with respect to traditional cache organizations is negligible in terms of hardware, as just simple changes in the circuitry of a CMP are required. DAPSCO achieves significant savings in both the execution time of applications and the energy consumption of the interconnection network when compared to the traditional partially shared cache organization in which clusters of the tiles share their LLC banks. We have simulated two examples of DAPSCO that improve the performance of a 64-core CMP by 4% and 6% with an underlying mesh topology, and by 10% and 13% with an underlying torus topology, all of it with respect to traditional partially shared caches with sharing degrees of 8 and 16. Network power consumption also gets reduced by 4% and 6% (mesh), and 10% and 13% (torus) regarding the same traditional configurations. We have also shown that as the number of cores and sharing degree increase, link traversals account for a growing fraction of execution time and the energy consumption of the whole chip, while at the same time DAPSCO removes a higher percentage of links from the critical path of cache misses, becoming even more effective (e.g., in a 512 CMP with a sharing degree of 128 and a torus, the latency to access the LLC can improve by 50%).

- **We have shown a novel way to store directory information more efficiently and with a simpler design by assigning the storage resources of the chip dynamically to either memory blocks or directory entries with a granularity of a cache entry (Chapter 5).** We call our new cache organization ICCI. ICCI allocates just the strictly necessary number of directory entries required at runtime, preventing the need for a dedicated structure sized to fit every worst-case scenario that eventually becomes unaffordable. ICCI takes advantage of the observation that more cores create more opportunities for data sharing, resulting in each tile typically storing fewer directory entries as the number of cores grows (because each directory entry tracks more private-cache blocks), making the relative resource usage for directory information in ICCI decrease as the number of cores rises. The analysis of the typical characteristics of workloads suggest high scalability for directory information in ICCI (with reported scenarios making ICCI take up less area for directory information than SCI or duplicate-tag directories with as few as 64 cores, without any of their drawbacks). ICCI outperforms other state-of-the-art proposals, especially in terms of energy. ICCI can use precise bit-vectors, although it is compatible with any other sharing code. Moreover, ICCI can be used in combination with elaborate sharing codes to apply it to extremely large core counts. Finally, the number of entries in the LLC is huge compared to separate directories, solving the problem of directory-induced invalidations in ICCI.

- **We have shown the importance and benefits of using adequate management policies to enable efficient use of hybrid photonic-electronic networks (Chapter 6).** We have designed and evaluated the first fine-grain policies that improve the throughput and latency of these hybrid networks with any amount of photonic resources, basing their decisions on distance between endpoints, ring availability and message size. The resulting network is more flexible and fairer to all cores, with each sub-network dedicated to transmitting those messages more suitable for the particular technology (e.g., long-distance transmissions through the photonic sub-network), balancing the message distribution dynamically at a message granularity depending on network load. We have tested these policies both on an affordable photonic ring for near-future CMPs and on large ring-based photonic networks, obtaining large performance improvements and energy consumption reductions in every case. When managed by an appropriate policy, the throughput of the hybrid network is higher than the sum of the throughputs of the sub-networks when working alone, and its latency is also low as a result of transmitting each message through the appropriate sub-network.

We expect that the work carried out in this thesis will help improve the scalability of future CMP designs. This thesis also opens several future ways of research. Among them, the following appear as the most promising:

- **Specific replacement policies for multiple-area cache coherence protocols**. These policies would take into account proximity to improve the operation of the provider mechanism. By replacing blocks in shared state before those for which the node acts as a provider, we can increase the overall utilization and accuracy of the prediction mechanism used to locate providers. Actively accessed providers would be reinforced by such a replacement mechanism, which would also reduce the amount of messages injected in the network (e.g., to relocate a provider and update prediction information) and the occurrence of coherence races that degrade performance.

- **Elaborate cache coherence protocols for increasing data proximity**. The fundamentals of multiple-area cache coherence protocols can be developed further by means of tree-based protocols (in which nodes recursively store sharing information about two or more children nodes) that take into account distance while building the tree. We have carried out studies that show that this kind of design has a great potential [57] in terms of area for the directory information, network traffic and latency, all of which combined can help boost energy efficiency noticeably. The key idea consists of inserting each new sharer in the branch with the closer child at each fork of the tree. This way, distances for messages between adjacent nodes of the tree decrease notably. By using prediction, as in our multiple-area proposals, and recording the parent in the tree as the provider, we can reach a very close provider upon cache misses (often physically adjacent in the chip), because thanks to the tree creation process, the parent is commonly a close node. Nodes using a block for long periods become fixed in the levels near the root of the tree, and those with more irregular usage of the block become leaves that retrieve data from the previous ones, ensuring both stability of the tree and good prediction accuracy. This mechanism can also be reinforced if necessary by proper replacement mechanisms. In addition, the tree provides natural broadcast for invalidation messages and recollection of responses.

- **Combination of multiple-area protocols and DAPSCO to design a new scheme embodying the best features of both**. For instance, a DAPSCO design can be used in the formation of the areas to reduce the average distance between its members compared to the original areas, increasing the benefits of using providers. Notice that it is possible to decouple the size of the areas (which we may want to make small to increase data proximity) and the sharing degree of the LLC (which we may want to make large to prevent an increase in LLC pressure caused by replicated data), to make each element benefit from a DAPSCO design in the most interesting way.

- **Study of ways to integrate ICCI with multiple-area protocols and DAPSCO**. By using ICCI to store directory information, we can improve the scalability of any directory-based cache coherence protocol. Using the sharing code distribution of multiple area coherence protocols we can achieve a different trade-off for ICCI in terms of scalability for very large core counts while also obtaining data proximity benefits. In addition, partially-shared caches are an interesting design choice whose interactions with ICCI are yet to be studied.

- **Dynamic private/shared caches with low storage usage for cache coherence**. ICCI provides natural support for adaptive private/shared last-level caches by dynamically storing the necessary amount of directory entries in the LLC. Private caches have drawbacks like limiting the cache resources available to each core to its local bank, replicating shared data and requiring a large directory to track the private cache contents. Shared caches have other shortcomings such as increasing network traffic and cache-hit latency. By using ICCI, the private/shared nature of the cache can be dynamically adjusted to fit application needs at an arbitrary granularity ranging from block level to bank level, by using ICCI's ability to allocate the strictly necessary number of directory entries in the LLC. This results in a framework that enables many policies. For instance, a node creating a NoC traffic bottleneck with excessive outgoing shared-cache accesses can alleviate it by replacing and retrieving blocks locally at the cost of increasing the pressure on its local LLC bank (and then do the opposite if its local LLC miss rate becomes excessive). If the traffic bottleneck is caused by incoming shared-cache requests to the node, that node can recommend requestors to replace the requested blocks locally, and in turn requestors can decide to obey depending on their current local bank miss rate. This would result in a more efficient cache behavior than private or shared caches in both energy and latency terms, with little additional complexity over ICCI. Already proposed mechanisms, such as Victim Replication [184], are straightforward to implement within the ICCI framework, without their original area overheads for directory information. In addition, by deactivating coherence for private blocks (Section 5.3.4.2), these do not increase pressure on the last level cache if they are replaced locally instead of to the home node, as no cache entry is required for the directory information in the home node. Other policies are possible, such as allowing a maximum number of blocks replaced locally (maybe dynamically adjustable). A single counter in each LLC bank suffices to implement this policy. The counter is incremented upon a local replacement and decremented upon a remote replacement from the local LLC bank to the home LLC bank. Also, more elaborate thresholds can be dynamically used to allow the local replacement of those blocks recurrently replaced showing great temporal locality in the LLC, obtaining the maximum benefits with the smallest LLC pressure increase.

- **Study of adapted management policies for hybrid networks to improve their efficiency when working in combination with different cache organizations and cache coherence protocols**. Management policies adapted to particular cache organizations (such as DAPSCO) could make a standard NoC adjust its operation to suit a wide range of chips, preventing the need to create specific photonic NoC layouts fitting the particular characteristics of each chip. Also, these policies would probably need to consider new parameters when adding ICCI (and the dynamic cache organizations that may follow from it) to the discussion.

Bibliography

[1] AMD Athlon 64 X2 Dual-Core Processor Product Data Sheet, 2007. 1.3, 4.1

[2] Manuel E. Acacio, José González, José M. García, and José Duato. A Two-Level Directory Architecture for Highly Scalable cc-NUMA Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 16:67–79, 2005. 1.2, 5.1

[3] S.V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996. 1, 1.2

[4] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture (ISCA)*, pages 280–298, 1988. 1.2, 3.1.1, 5.1

[5] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K. Jha. GARNET: A Detailed On-Chip Network Model Inside a Full-System Simulator. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 33–42, 2009. 3.4.1, 4.3.2

[6] Niket Agarwal, Li-Shiuan Peh, and Niraj K Jha. In-network Coherence Filtering: Snoopy Coherence Without Broadcasts. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 232–243, 2009. 5.3.4.1, 5.3.4.1

[7] Rajesh Agny, Eric DeLano, Mohan Kumar, Murugasamy Nachimuthu, and Robert Shiveley. The Intel Itanium Processor 9300 Series. Intel White Paper, 2010. 1.3, 4.1

[8] C. Auth, C. Allen, A. Blattner, D. Bergstrom, M. Brazier, M. Bost, M. Buehler, V. Chikarmane, T. Ghani, T. Glassman, R. Grover, W. Han, D. Hanken,

M. Hattendorf, P. Hentges, R. Heussner, J. Hicks, D. Ingerly, P. Jain, S. Jaloviar, R. James, D. Jones, J. Jopling, S. Joshi, C. Kenyon, H. Liu, R. McFadden, B. McIntyre, J. Neirynck, C. Parker, L. Pipes, I. Post, S. Pradhan, M. Prince, S. Ramey, T. Reynolds, J. Roesler, J. Sandford, J. Seiple, P. Smith, C. Thomas, D. Towner, T. Troeger, C. Weber, P. Yashar, K. Zawadzki, and K. Mistry. A 22nm high performance and low-power CMOS technology featuring fully-depleted tri-gate transistors, self-aligned contacts and high density MIM capacitors. In *2012 Symposium on VLSI Technology (VLSIT)*, pages 131–132, 2012. 1.1

[9] J. L. Baer and W. H. Wang. On the inclusion properties for multi-level cache hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer Architecture (ISCA)*, pages 73–80, 1988. 1.2, 5.1.2

[10] Shirish Bahirat and Sudeep Pasricha. A particle swarm optimization approach for synthesizing application-specific hybrid photonic networks-on-chip. In *13th International Symposium on Quality Electronic Design (ISQED)*, pages 78–83. IEEE, 2012. 6.1

[11] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M Balakrishnan, and Peter Marwedel. Scratchpad Memory: Design Alternative for Cache On-Chip Memory in Embedded Systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, pages 73–78, 2002. 1.2

[12] Luiz André Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th annual International Symposium on Computer Architecture (ISCA)*, pages 282–293, 2000. 1.2, 5.1, 5.1.4

[13] Nick Barrow-Williams, Christian Fensch, and Simon Moore. Proximity coherence for chip multiprocessors. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT)*, pages 123–134, 2010. 3.4.1

[14] John Barth, Don Plass, Erik Nelson, Charlie Hwang, Gregory Fredeman, Michael Sperling, Abraham Mathews, William Reohr, Kavita Nair, and Nianzheng Cao. A 45nm SOI embedded DRAM macro for POWER7™ 32MB on-chip L3 cache. In *2010 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 342–343, 2010. 1.3

[15] S. Bartolini and P. Grani. A Simple On-Chip Optical Interconnection for Improving Performance of Coherency Traffic in CMPs. In *15th Euromicro Conference on Digital System Design (DSD)*, pages 312–318, 2012. 6.4

[16] Bradford M. Beckmann, Michael R. Marty, and David A. Wood. ASR: Adaptive Selective Replication for CMP Caches. *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 443–454, 2006. 4.4, 5.3.4.1, 5.3.4.1

[17] Bradford M. Beckmann and David A. Wood. Managing wire delay in large chip-multiprocessor caches. In *37th International Symposium on Microarchitecture (MICRO)*, pages 319–330, 2004. 5.3.4.1

[18] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64-Processor: A 64-Core SoC with Mesh Interconnect. In *IEEE International Solid-State Circuits Conference (ISSCC)*, pages 88–598, 2008. 1.4

[19] Luca Benini and Giovanni De Micheli. Networks on Chips: A new SoC paradigm. *Computer*, 35(1):70–78, 2002. 1.3

[20] Dileep Bhandarkar and Jason Ding. Performance Characterization of the Pentium Pro Processor. In *Third International Symposium on High-Performance Computer Architecture (HPCA)*, pages 288–297, 1997. 1.1

[21] Christian Bienia and Kai Li. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2009. 6.1, 6.3.1.1

[22] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, August 2011. 6.3.1.1

[23] Matthias A. Blumrich and Valentina Salapura. Programmable partitioning for high-performance coherence domains in a multiprocessor system. United States Patent No. US 2009/0006769 A1, 2009. International Business Machines Corporation. 3.1.2, 3.5

[24] Wim Bogaerts, Peter De Heyn, Thomas Van Vaerenbergh, Katrien De Vos, Shankar Kumar Selvaraja, Tom Claes, Pieter Dumon, Peter Bienstman, Dries Van Thourhout, and Roel Baets. Silicon Microring Resonators. *Laser & Photonics Reviews*, 6(1):47–73, 2012. 2.2

[25] Mark Bohr and Kaizad Mistry. Intel's revolutionary 22 nm transistor technology. *Intel Corporation*, 2011. 1.1

[26] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference (DAC)*, pages 746–749, 2007. 1.1

[27] Shekhar Borkar and Andrew A. Chien. The Future of Microprocessors. *Communications of the ACM*, 54(5):67–77, May 2011. 1, 1.3, 6.1

[28] Nathan Brookwood. AMD Fusion Family of APUs: Enabling a Superior, Immersive PC Experience. *Insight*, 64(1):1–8, 2010. 1.4

[29] Ray Bryant, J. Hawkes, J. Steiner, J. Barnes, and J. Higdon. Scaling Linux to the Extreme. In *Proceedings of the Linux Symposium*, pages 133–148, 2004. 1.2

[30] Brian Case. Intel Reveals Pentium Implementation Details. *Microprocessor Report*, 5(23):9–17, 1993. 1.1

[31] L.M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, dec. 1978. 1.2, 5.1

[32] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Int. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 224–234, 1991. 3.1.1

[33] Jichuan Chang and Gurindar S. Sohi. Cooperative Caching for Chip Multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*, pages 264–276, 2006. 4.4

[34] Mainak Chaudhuri, Jayesh Gaur, Nithiyanandan Bashyam, Sreenivas Subramoney, and Joseph Nuzman. Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches. In *Proceedings of the 21st international conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 293–304, 2012. 1.2, 5.1.2

[35] Liqun Cheng, Naveen Muralimanohar, Karthik Ramani, Rajeev Balasubramonian, and John B. Carter. Interconnect-aware coherence protocols for chip multiprocessors. In *Proceedings of the 33rd annual international symposium on Computer Architecture (ISCA)*, pages 339–351, 2006. 6.1

[36] Sangyeun Cho and Lei Jin. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In *IEEE/ACM International Symposium on Michroarchitecture (MICRO)*, pages 455–468, 2006. 4.4

[37] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *IEEE Micro*, 30(2):16–29, March 2010. 1.4, 5.1.3, 5.2.2, 5.2.3, 5.3.1

[38] John H. Crawford. The i486 CPU: Executing Instructions in One Clock Cycle. *IEEE Micro*, 10(1):27–36, 1990. 1.1

[39] Blas Cuesta, Alberto Ros, Maria E. Gómez, Antonio Robles, and Jose Duato. Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks. In *38th Annual International Symposium on Computer Architecture (ISCA)*, pages 93–103, 2011. 5.1, 5.3.4.1, 5.3.4.2

[40] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science Engineering*, 5(1):46–55, 1998. 1.2

[41] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. CPU DB: Recording Microprocessor History. *ACM Queue*, 10(4):10–27, 2012. 1

[42] Suman Datta. Recent Advances in High Performance CMOS Transistors: From Planar to Non-Planar. *Electrochemical Society Interface*, pages 41–46, 2013. 1.1

[43] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. Leblanc. Design of ion-implanted MOSFETs with very small physical dimensions. *IEEE Journal of Solid-state Circuits*, SC-9(5):256–268, Oct. 1974. 1, 1.1

[44] Jose Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection Networks: An Engineering Approach.* Morgan Kaufmann, 2002. 3.4.1

[45] Natalie D. Enright Jerger, Li-Shiuan Peh, and Mikko H. Lipasti. Virtual Tree Coherence: Leveraging Regions and In-Network Multicast Trees for Scalable Cache Coherence. In *41st IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 35–46, 2008. 5.3.4.1, 5.3.4.1

[46] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Power challenges may end the multicore era. *Communications of the ACM*, 56(2):93–102, February 2013. 1.1

[47] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011. 1, 1.1

[48] F. Faggin, M.E. Hoff, S. Mazor, and M. Shima. The history of the 4004. *IEEE Micro*, 16(6):10–20, 1996. 1.1

[49] Federico Faggin. The Making of the First Microprocessor. *IEEE Solid-State Circuits Magazine*, 1(1):8–21, 2009. 1

[50] Chris Fallin, Greg Nazario, Xiangyao Yu, Kevin Chang, Rachata Ausavarungnirun, and Onur Mutlu. MinBD: Minimally-Buffered Deflection Routing for Energy-Efficient Interconnect. In *Proceedings of the 2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip (NOCS)*, NOCS '12, pages 1–10, 2012. 6.3.2

[51] Michael Ferdman, Pejman Lotfi-Kamran, Ken Balet, and Babak Falsafi. Cuckoo Directory: A Scalable Directory for Many-Core Systems. In *17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 169–180, 2011. 1.2, 5.1, 5.1.3, 5.3.4.1, 5.3.4.1, 5.4

[52] T. Fischer, S. Arekapudi, E. Busta, C. Dietz, M. Golden, S. Hilker, A. Horiuchi, K.A. Hurd, D. Johnson, H. McIntyre, S. Naffziger, J. Vinh, J. White, and K. Wilcox. Design solutions for the Bulldozer 32nm SOI 2-core processor module in an 8-core CPU. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 78–80, 2011. 1.4

[53] Jose Flich, Jose Duato, Thomas Sødring, Åshild Grønstad Solheim, Tor Skeie, Olav Lysne, and Samuel Rodrigo. On the Potential of NoC Virtualization for Multicore Chips. In *International Workshop on Multi-Core Computing Systems (MuCoCoS)*, pages 801–807, 2008. 3.5

[54] Antonio Flores, Juan L. Aragón, and Manuel E. Acacio. Heterogeneous Interconnects for Energy-Efficient Message Management in CMPs. *IEEE Transactions on Computers*, 59(1):16–28, 2010. 3.5, 4.4, 6.1

[55] Samuel H. Fuller and Lynette I. Millett. *The Future of Computing Performance: Game Over or Next Level?* National Academies Press, 2011. 1.1

[56] Antonio García-Guirado, Ricardo Fernández-Pascual, and José M. García. Virtual-GEMS: An Infrastructure To Simulate Virtual Machines. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, pages 53–62, 2009. 1.7, 3.4.1, 4.3.2

[57] Antonio García-Guirado, Ricardo Fernández-Pascual, and José M. García. Coherencia de Caché Mediante Árbol Basado en Proximidad y Predicción. In *XXII Jornadas de Paralelismo*, pages 239–244, 2011. 7

[58] Antonio García-Guirado, Ricardo Fernández-Pascual, and José M. García. In-Cache Coherence Information. *Under review*. 1.7

[59] Antonio García-Guirado, Ricardo Fernández-Pascual, and José M. García. On the Virtual Hierarchy Cache Coherence Protocols for Server Consolidation. *Under review*. 1.7

[60] Antonio García-Guirado, Ricardo Fernández-Pascual, José M. García, and Sandro Bartolini. Dynamic Management Policies for Exploiting Hybrid Photonic-Electronic NoCs. *Under review*. 1.7

[61] Antonio García-Guirado, Ricardo Fernández-Pascual, and José M. García. Analyzing Cache Coherence Protocols for Server Consolidation. In *Proceedings of the 22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 191–198, 2010. 1.7, 3.1, 3.5

[62] Antonio García-Guirado, Ricardo Fernández-Pascual, Alberto Ros, and José M. García. Energy-efficient cache coherence protocols in chip-multiprocessors for server consolidation. In *Proceedings of the 40th International Conference on Parallel Processing (ICPP)*, pages 51–62, 2011. 1.7

[63] Antonio García-Guirado, Ricardo Fernández-Pascual, Alberto Ros, and José M. García. DAPSCO: Distance-Aware Partially Shared Cache Organization. *ACM Trans. Archit. Code Optim.*, 8(4):25:1–25:19, January 2012. 1.7, 6.2.3

[64] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005. 1, 1.1

[65] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th annual International Symposium on Computer Architecture (ISCA)*, pages 124–131, 1983. 1.1, 1.2, 5.1

[66] Ed Grochowski and Murali Annavaram. Energy per Instruction Trends in Intel Microprocessors. *Technology at Intel Magazine*, 4(3):1–8, 2006. 1.1

[67] William Gropp, Ewing L. Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT press, 1999. 1.2

[68] Cary Gunn. CMOS Photonics for High-Speed Interconnects. *IEEE Micro*, 26(2):58–66, March 2006. 1.3, 6.1

[69] A. Gupta, W.D. Weber, and T. Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *Int. Conference on Parallel Processing (ICPP)*, pages 312–321, 1990. 1.2, 3.1.1, 4.1.2, 5.1, 5.1, 5.1.3

[70] E.P. Gusev, E. Cartier, D.A. Buchanan, M. Gribelyuk, M. Copel, H. Okorn-Schmidt, and C. D'Emic. Ultrathin high-K metal oxides on silicon: processing, characterization and integration issues. *Microelectronic Engineering*, 59(1-4):341–349, 2001. 1.1

[71] Zvika Guz, Idit Keidar, Avinoam Kolodny, and Uri C. Weiser. Nahalal: Memory Organization for Chip Multiprocessors. Technical report, Technion-IIT, Department of Electrical Engineering, 2006. 5.3.4.1, 5.3.4.1

[72] Zvika Guz, Idit Keidar, Avinoam Kolodny, and Uri C Weiser. Utilizing Shared Data in Chip Multiprocessors with the Nahalal Architecture. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 1–10, 2008. 5.3.4.1

[73] Linley Gwennap. Sandy Bridge Spans Generations. *Microprocessor Report*, 9(27):1–8, 2010. 1.4

[74] W. Haensch, E.J. Nowak, R.H. Dennard, P.M. Solomon, A. Bryant, O.H. Dokumaci, A. Kumar, X. Wang, J.B. Johnson, and M.V. Fischetti. Silicon CMOS devices beyond scaling. *IBM Journal of Research and Development*, 50(4.5):339–361, 2006. 1, 1.1

[75] Mohammad Hammoud, Sangyeun Cho, and Rami Melhem. Dynamic Cache Clustering for Chip Multiprocessors. In *Proceedings of the 23rd International Conference on Supercomputing (ICS)*, pages 56–67, 2009. 1.3, 4.1, 4.4

[76] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. In *Proceedings of the 36th Annual International Symposium on Computer architecture (ISCA)*, pages 184–195, 2009. 4.3.4, 4.4, 5.3.4.2

[77] A. Hartstein, V. Srinivasan, T. R. Puzak, and P. G. Emma. Cache miss behavior: is it $\sqrt{2}$. In *Proceedings of the 3rd conference on Computing frontiers (CF)*, pages 313–320, 2006. 5.3.2.1

[78] Gang He, Zhaoqi Sun, Mao Liu, and Lide Zhang. Scaling and Limitation of Si-Based CMOS. *High-k Gate Dielectrics for CMOS Technology*, pages 1–29, 2012. 1.1

[79] Jim Held, Jerry Bautista, and Sean Koehl. From a Few Cores to Many: A Tera-scale Computing Research Overview. Intel White Paper, 2006. 1.3

[80] Jim Held and Sean Koehl. Inside Intel Core Microarchitecture. Intel White Paper, 2006. 1.3, 4.1

[81] Jim Held and Sean Koehl. Introducing the Single-Chip Cloud Computer. Intel White Paper, 2010. 1.4, 5.3.1

[82] Gilbert Hendry, Shoaib Kamil, Aleksandr Biberman, Johnnie Chan, Benjamin G. Lee, Marghoob Mohiyuddin, Ankit Jain, Keren Bergman, Luca P. Carloni, John Kubiatowicz, Leonid Oliker, and John Shalf. Analysis of photonic networks for a chip multiprocessor using scientific applications. In *Proceedings of the 2009 3rd ACM/IEEE International Symposium on Networks-on-Chip (NOCS)*, pages 104–113, Washington, DC, USA, 2009. IEEE Computer Society. 6.1

[83] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, 1:1–13, 2001. 1.1

[84] Yatin Hoskote, Sriram Vangal, Arvind Singh, Nitin Borkar, and Shekhar Borkar. A 5-GHz Mesh Interconnect for a Teraflops Processor. *IEEE Micro*, 27(5):51–61, 2007. 1.3

[85] Min Huang, Moty Mehalel, Ramesh Arvapalli, and Songnian He. An Energy Efficient 32nm 20 MB L3 Cache for Intel® Xeon® Processor E5 Family. In *2012 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–4, 2012. 1.3

[86] Christopher Hughes, Changkyu Kim, and Yen-Kuang Chen. Performance and Energy Implications of Many-Core Caches for Throughput Computing. *IEEE Micro*, 30(6):25–35, 2010. 4.4

[87] Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W. Keckler. A NUCA Substrate for Flexible CMP Cache Sharing. In *Proceedings of the 19th annual international conference on Supercomputing (ICS)*, pages 31–40, 2005. 1.3, 1.6.2, 4.1, 4.4

[88] Adrian M. Ionescu and Heike Riel. Tunnel field-effect transistors as energy-efficient electronic switches. *Nature*, 479(7373):329–337, 2011. 1.1

[89] ITRS. International technology roadmap for semiconductors, the 2012 update. http://www.itrs.net (2012). 1.1

[90] B. Jalali and S. Fathpour. Silicon photonics. *Lightwave Technology, Journal of*, 24(12):4600–4615, dec. 2006. 1.3, 6.1

[91] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. Achieving Non-Inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 151–162, 2010. 5.2.1, 5.3.1, 5.3.2.1

[92] David V James, Anthony T Laundrie, Stein Gjessing, and Gurindar S Sohi. Distributed-Directory Scheme: Scalable Coherent Interface. *IEEE Computer*, 23(6):74–77, 1990. 5.1

[93] Myeongjae Jeon, Euiseong Seo, Junghyun Kim, and Joonwoon Lee. Domain Level Page Sharing in Xen Virtual Machine Systems. In *The 7th International Symposium on Advanced Parallel Processing Technologies (APPT)*, pages 590–599, 2007. 3.1

[94] Andrew B. Kahng, Bin Li, Li-Shiuan Peh, and Kambiz Samadi. ORION 2.0: a Fast and Accurate NoC Power and Area Model for Early-Stage Design Space Exploration. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 423–428, 2009. 4.3.2, 6.3.1.1

[95] Ron Kalla, Balaram Sinharoy, William J. Starke, and Michael Floyd. POWER7: IBM's Next-Generation Server Processor. *IEEE Micro*, 30(2):7–15, 2010. 1.3, 4.1

[96] Stefanos Kaxiras and Margaret Martonosi. Optimizing capacitance and switching activity to reduce dynamic power. *Computer Architecture Techniques for Power-Efficiency*, pages 45–129, 2008. 1.1

[97] John H. Kelm, Matthew R. Johnson, Steven S. Lumettta, and Sanjay J. Patel. WAYPOINT: scaling coherence to thousand-core architectures. In *Proceedings of the 19th international conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 99–110, 2010. 1.2, 5.1, 5.4

[98] Georgios Keramidas and Stefanos Kaxiras. SARC Coherence: Scaling Directory Cache Coherence in Performance and Power (preprint). *IEEE Micro*, 2010. 3.5

[99] Kurt Keutzer, Sharad Malik, and A. Richard Newton. From ASIC to ASIP: The Next Design Discontinuity. In *Proceedings of 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 84–90, 2002. 1.4

[100] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 211–222, 2002. 1.3, 4.1, 4.4, 5.1.1, 6.1

[101] Ryan C. Kinter. Support for multiple coherence domains. Patent No. WO 2009/039417 A1, 2009. MIPS Technologies, Inc. 3.1.2, 3.5

[102] Scott Kirkpatrick, D. Gelatt Jr., and Mario P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983. 4.2.1

[103] Nevin Kirman, Meyrem Kirman, Rajeev K. Dokania, Jose F. Martinez, Alyssa B. Apsel, Matthew A. Watkins, and David H. Albonesi. Leveraging optical technology in future bus-based chip multiprocessors. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 492–503, 2006. 6.1.1

[104] Peter M. Kogge. *The Architecture of Pipelined Computers*. Taylor & Francis Group, 1981. 1.1

[105] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, 2005. 1.3, 4.1, 4.4

[106] J.G. Koomey, S. Berard, M. Sanchez, and H. Wong. Implications of Historical Trends in the Electrical Efficiency of Computing. *IEEE Annals of the History of Computing*, 33(3):46–54, 2011. 1

[107] Anil Krishna, Ahmad Samih, and Yan Solihin. Data sharing in multithreaded applications and its impact on chip design. In *2012 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 125–134, 2012. 5.3.4.1

[108] Snehasish Kumar, Hongzhou Zhao, Arrvindh Shriraman, Eric Matthews, Sandhya Dwarkadas, and Lesley Shannon. Amoeba-Cache: Adaptive Blocks for Eliminating Waste in the Memory Hierarchy. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 376–388, 2012. 5.3.4.2

[109] George Kurian, Jason E. Miller, James Psota, Jonathan Eastep, Jifeng Liu, Jurgen Michel, Lionel C. Kimerling, and Anant Agarwal. ATAC: a 1000-core cache-coherent processor with on-chip optical network. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT)*, pages 477–488, 2010. 5.3.4.1, 6.4

[110] James Laudon and Daniel Lenoski. The SGI Origin: a ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA)*, pages 241–251, 1997. 1.2, 5.1

[111] Todd Legler. Choosing the DRAM with Complex System Considerations. Presented at Embedded Systems Conference 2012, March 2012. 5.3.1

[112] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–480, 2009. 5.3.1

[113] Yong Li, Rami Melhem, and Alex K Jones. Practically private: Enabling high performance cmps through compiler-assisted data classification. In

Proceedings of the 21st international conference on Parallel Architectures and Compilation Techniques (PACT), pages 231–240, 2012. 5.1, 5.3.4.2

[114] Zheng Li, Dan Fay, Alan Rolf Mickelson, Li Shang, Manish Vachharajani, Dejan Filipovic, Wounjhang Park, and Yihe Sun. Spectrum: a hybrid nanophotonic-electric on-chip network. In *46th Annual Design Automation Conference (DAC)*, pages 575–580, 2009. 6.1

[115] Chun Liu, Anand Sivasubramaniam, and Mahmut Kandemir. Organizing the Last Line of Defense before Hitting the Memory Wall for CMPs. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA)*, pages 176–185, 2004. 1.2, 1.3, 4.1

[116] Jack L. Lo, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, Dean M. Tullsen, and S. J. Eggers. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15(3):322–354, 1997. 1.1

[117] Pejman Lotfi-Kamran, Michael Ferdman, Daniel Crisan, and Babak Falsafi. TurboTag: Lookup Filtering to Reduce Coherence Directory Power. In *Proceedings of the 16th International Symposium on Low Power Electronics and Design (ISLPED)*, pages 377–382, 2010. 3.5, 4.4

[118] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, 2005. 5.3

[119] S. Lutkemeier, T. Jungeblut, H.K.O. Berge, S. Aunet, M. Porrmann, and U. Ruckert. A 65 nm 32 b Subthreshold Processor With 9T Multi-Vt SRAM and Adaptive Supply Voltage Control. *Journal of Solid-State Circuits*, 48(1):8–19, 2013. 1.5

[120] Sheng Ma, Natalie Enright Jerger, and Zhiying Wang. Supporting efficient collective communication in NoCs. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12, 2012. 5.3.1

[121] Nir Magen and Avinoam Kolodny. Interconnect-Power Dissipation in a Microprocessor. In *Proceedings of the International Workshop on System-Level Interconnect Prediction*, pages 7–13, 2004. 3.1

[122] Nir Magen, Avinoam Kolodny, Uri Weiser, and Nachum Shamir. Interconnect-power dissipation in a microprocessor. In *Int'l workshop on System Level Interconnect Prediction (SLIP)*, pages 7–13, February 2004. 6.1

[123] Turbo Majumder, Partha Pratim Pande, and Ananth Kalyanaraman. High-throughput, energy-efficient network-on-chip-based hardware accelerators. *Sustainable Computing: Informatics and Systems*, 3(1):36–46, 2013. 6.4

[124] Dan C. Marinescu. *Cloud Computing: Theory and Practice*. Morgan Kaufmann, 2013. 3.1

[125] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why on-chip cache coherence is here to stay. *Communications of the ACM*, 55(7):78–89, July 2012. 1, 1.2, 4.1.2, 5.1, 5.1, 5.1.4, 6.1.1, 6.2.1

[126] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005. 4.3.2, 5.3

[127] Milo M.K. Martin, Pacia J. Harper, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors. In *30th Annual International Symposium on Computer Architecture (ISCA)*, pages 206–217. 5.3.4.1

[128] Michael R. Marty and Mark D. Hill. Virtual Hierarchies to Support Server Consolidation. In *Proceedings of the 34th annual international symposium on Computer architecture (ISCA)*, pages 46–56, 2007. 3.1.2, 3.5

[129] Marvin Minsky. Steps Toward Artificial Intelligence. *Proceedings of the IRE*, 49(1):8–30, 1961. 4.2.1

[130] D. Molka, D. Hackenberg, R. Schone, and M.S. Muller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 261–270, 2009. 5.1.2

[131] Matteo Monchiero, Jung Ho Ahn, Ayose Falcón, Daniel Ortega, and Paolo Faraboschi. How to simulate 1000 cores. *SIGARCH Comput. Archit. News*, 37(2):10–19, July 2009. 5.3

[132] Gordon E. Moore. Cramming more Components onto Integrated Circuits. *Electronics*, 38(8):114–117, 1965. 1.1

[133] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTI 6.0: A Tool to Model Large Caches. Technical report, Hewlett Packard, 2009. 3.4.1

[134] Richard C. Murphy and Peter M. Kogge. On the memory access patterns of supercomputer applications: Benchmark selection and its implications. *IEEE Transactions on Computers*, 56(7):937–945, 2007. 3.4.3

[135] B. A. Nayfeh, K. Olukotun, and J. P. Singh. The Impact of Shared-Cache Clustering in Small-Scale Shared-Memory Multiprocessors. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 74–84, 1996. 1.3, 4.1

[136] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads programming - a POSIX standard for better multiprocessing*. O'Reilly, 1996. 1.2

[137] Håkan Nilsson and Per Stenstrom. The scalable tree protocol-a cache coherence approach for large-scale multiprocessors. In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, pages 498–506, 1992. 5.1

[138] Christopher Nitta, Matthew Farrens, and Venkatesh Akella. DCAF - A Directly Connected Arbitration-Free Photonic Crossbar For Energy-Efficient High Performance Computing. In *26th International Parallel & Distributed Processing Symposium (IPDPS)*, pages 1–12, 2012. 6.4

[139] Ian O'Connor, Dries Van Thourhout, and Alberto Scandurra. Wavelength division multiplexed photonic layer on CMOS. In *Proceedings of the 2012 Interconnection Network Architecture: On-Chip, Multi-Chip Workshop*, pages 33–36, 2012. 6.1.1

[140] Zeev Offen, Ariel Berkovits, and Piazza Thomas. Technique to share information among different cache coherency domains. Patent No. WO 2009/120997 A2, 2009. Intel Corporation. 3.1.2, 3.5

[141] Tae Cheol Oh. *Analytical Models for Chip Multiprocessor Memory Hierarchy Design and Management*. PhD in Computer science, University of Pittsburgh - Pennsylvania, 2010. 5.3.4.1, 5.3.4.1

[142] Taecheol Oh, Hyunjin Lee, Kiyeon Lee, and Sangyeun Cho. An analytical model to study optimal area breakdown between cores and caches in a chip multiprocessor. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 181–186, 2009. 5.3.4.1

[143] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The Case for a Single-Chip Multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–11, 1996. 1.1

[144] John D Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008. 1.4

[145] Yan Pan, John Kim, and Gokhan Memik. Flexishare: Channel sharing for an energy-efficient nanophotonic crossbar. In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12. IEEE Computer Society, 2010. 6.1, 6.1.1, 6.1.2.1, 6.3, 6.3.1, 6.3.1.1, 6.4

[146] Yan Pan, John Kim, and Gokhan Memik. FeatherWeight: low-cost optical arbitration with QoS support. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 105–116, 2011. 6.4

[147] Yan Pan, Prabhat Kumar, John Kim, Gokhan Memik, Yu Zhang, and Alok Choudhary. Firefly: Illuminating future network-on-chip with nanophotonics. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 429–440, 2009. 6.1, 6.1.1, 6.1.2.3, 6.2.4, 6.3, 6.3.2, 6.3.4

[148] M. Petracca, B.G. Lee, K. Bergman, and L.P. Carloni. Design exploration of optical interconnection networks for chip multiprocessors. In *16th IEEE Symposium on High Performance Interconnects*, pages 31–40, 2008. 6.1.1, 6.4

[149] Tom Piazza, Hong Jiang, Per Hammarlund, and Ronak Singhal. Technology Insight: Intel® Next Generation Microarchitecture Code Name Haswell. Presented at Intel Developer Forum 2012, September 2012. 1.4

[150] Fred J. Pollack. New microarchitecture challenges in the coming generations of CMOS process technologies (keynote address). In *Proceedings of the 32nd*

annual ACM/IEEE international symposium on Microarchitecture, pages 2–, 1999. 1.1

[151] Seth H. Pugsley, Josef B. Spjut, David W. Nellans, and Rajeev Balasubramonian. SWEL: Hardware cache coherence protocols to map shared data onto shared caches. In *Proceedings of the 19th international conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 465–476. ACM, 2010. 5.3.4.1

[152] Carl Ramey. TILE-Gx100 ManyCore Processor: Acceleration Interfaces and Architecture. Presented at Hot Chips, August 2011. 1.3, 1.4

[153] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-Level Parallel Processing: History, Overview, and Perspective. *Journal of Supercomputing*, 7(1-2):9–50, May 1993. 1

[154] Brian M. Rogers, Anil Krishna, Gordon B. Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, pages 371–382, 2009. 1.3, 4.1, 5.3.4.1

[155] Alberto Ros, Manuel E. Acacio, and José M. García. A Direct Coherence Protocol for Many-Core Chip Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 21(12):1779–1792, 2010. 3.1, 3.1.2, 3.3, 3.3.1

[156] Alberto Ros, Marcelo Cintra, Manuel E. Acacio, and José M. García. Distance-Aware Round-Robin Mapping for Large NUCA Caches. In *Proceedings of the 16th International Conference on High Performance Computing (HiPC)*, pages 79–88, 2009. 4.4

[157] Praveen Salihundam, Shailendra Jain, Tiju Jacob, Shasi Kumar, Vasantha Erraguntla, Yatin Hoskote, Sriram Vangal, Gregory Ruhl, Partha Kundu, and Nitin Borkar. A 2Tb/s 6× 4 Mesh Network with DVFS and 2.3 Tb/s/W router in 45nm CMOS. In *2010 IEEE Symposium on VLSI Circuits (VLSIC)*, pages 79–80, 2010. 1.3

[158] Daniel Sanchez and Christos Kozyrakis. The zcache: Decoupling ways and associativity. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 187–198, 2010. 5.1.4

[159] Daniel Sanchez and Christos Kozyrakis. SCD: A scalable coherence directory with flexible sharer set encoding. In *Proceedings of the 18th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 129–140, 2012. 1.6.3, 5.1, 5.1.4, 5.2.3.1, 5.3.1, 5.3.4.1, 5.4

[160] Assaf Shacham, Keren Bergman, and Luca P. Carloni. Photonic networks-on-chip for future generations of chip multiprocessors. *IEEE Transactions on Computers*, 57(9):1246–1260, September 2008. 6.1.1, 6.4

[161] M. Shah, J. Barren, J. Brooks, R. Golla, G. Grohoski, N. Gura, R. Hetherington, P. Jordan, M. Luttrell, C. Olson, B. Sana, D. Sheahan, L. Spracklen, and A. Wynn. UltraSPARC T2: A Highly-Threaded, Power-Efficient, SPARC SoC. In *IEEE Asian Solid-State Circuits Conference*, pages 22–25, 2007. 4.4

[162] G.G. Shahidi. SOI technology for the GHz era. *IBM Journal of Research and Development*, 46(2.3):121–131, 2002. 1.1

[163] Ken Shoemaker. The i486 Microprocessor Integrated Cache and Bus Interface. In *Thirty-Fifth IEEE Computer Society International Conference Compcon Spring'90*, pages 248–253, 1990. 1.1

[164] James E. Smith and Gurindar S. Sohi. The Microarchitecture of Superscalar Processors. *Proceedings of the IEEE*, 83(12):1609–1624, 1995. 1.1

[165] Kyomin Sohn, Taesik Na, Indal Song, Yong Shim, Wonil Bae, Sanghee Kang, Dongsu Lee, Hangyun Jung, Hanki Jeoung, Ki-Won Lee, Junsuk Park, Jongeun Lee, Byunghyun Lee, Inwoo Jun, Juseop Park, Junghwan Park, Hundai Choi, Sanghee Kim, Haeyoung Chung, Young Choi, Dae-Hee Jung, Jang Seok Choi, Byungsick Moon, Jung-Hwan Choi, Byungchul Kim, Seong-Jin Jang, Joo Sun Choi, and Kyung Seok Oh. A 1.2V 30nm 3.2Gb/s/pin 4Gb DDR4 SDRAM with dual-error detection and PVT-tolerant data-fetch scheme. In *IEEE International Solid-State Circuits Conference (ISSCC)*, pages 38–40, 2012. 5.3.1

[166] Daniel J. Sorin, Mark D. Hill, and David A. Wood. A Primer on Memory Consistency and Cache Coherence. *Synthesis Lectures on Computer Architecture*, 2011. 1.2, 2.1, 2.1

[167] S.E. Thompson, M. Armstrong, C. Auth, M. Alavi, M. Buehler, R. Chau, S. Cea, T. Ghani, G. Glass, T. Hoffman, Chia-Hong Jan, C. Kenyon, J. Klaus, K. Kuhn, Zhiyong Ma, B. McIntyre, K. Mistry, A. Murthy, B. Obradovic,

R. Nagisetty, Phi Nguyen, S. Sivakumar, R. Shaheed, L. Shifren, B. Tufts, S. Tyagi, M. Bohr, and Y. El-Mansy. A 90-nm logic technology featuring strained-silicon. *IEEE Transactions on Electron Devices*, 51(11):1790–1797, 2004. 1.1

[168] Greg Thorson and Michael Woodacre. SGI® UV2: a fused computation and data analysis machine. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 105:1–105:9, 2012. 5.1, 5.4

[169] Robert M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33, 1967. 1.1

[170] Jurg Van Vliet and Flavia Paganelli. *Programming Amazon EC2*. O'Reilly Media, 2011. 3.1

[171] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-tile 1.28 TFLOPS Network-on-Chip in 65nm CMOS. In *IEEE International Solid-State Circuits Conference (ISSCC)*, pages 98–589, 2007. 1.3, 1.4

[172] D. Vantrease, N. Binkert, R. Schreiber, and M.H. Lipasti. Light speed arbitration and flow control for nanophotonic interconnects. In *Proceedings of the 42th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 304–315, 2009. 6.1.1, 6.4

[173] Dana Vantrease, Robert Schreiber, Matteo Monchiero, Moray McLaren, Norman P. Jouppi, Marco Fiorentino, Al Davis, Nathan Binkert, Raymond G. Beausoleil, and Jung Ho Ahn. Corona: System implications of emerging nanophotonic technology. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*, pages 153–164, 2008. 6.1, 6.1.1, 6.1.2.2, 6.3, 6.3.2, 6.4

[174] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *5th Symposium on Operating System Design and Implementation (OSDI)*, pages 181–194, 2002. 3.1

[175] David Wall. Limits of instruction-level parallelism. In *Proceedings of the fourth international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 176–188, 1991. 1.1

[176] Chunhua Xiao, M-C. Frank Chang, Jason Cong, Michael Gill, Zhangqin Huang, Chunyue Liu, Glenn Reinman, and Hao Wu. Stream arbitration: Towards efficient bandwidth utilization for emerging on-chip interconnects. *ACM Trans. Archit. Code Optim.*, 9(4), January 2013. 6.1.2.1

[177] Qianfan Xu, David Fattal, and Raymond G. Beausoleil. Silicon microring resonators with 1.5-μm radius. *Opt. Express*, 16(6):4309–4315, 2008. 2.2

[178] Yi Xu, Yu Du, Youtao Zhang, and Jun Yang. A composite and scalable cache coherence protocol for large scale CMPs. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 285–294, 2011. 6.1, 6.1.1, 6.4

[179] Tse-Yu Yeh and Yale N. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. In *Proceedings of the 25th annual international symposium on Microarchitecture (MICRO)*, pages 129–139, 1992. 1.1

[180] Jason Zebchuk, Vijayalakshmi Srinivasan, Moinuddin K. Qureshi, and Andreas Moshovos. A Tagless Coherence Directory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 423–434, 2009. 3.5, 5.1, 5.4

[181] Bo Zhai, Sanjay Pant, Leyla Nazhandali, Scott Hanson, Javin Olson, Anna Reeves, Michael Minuth, Ryan Helfand, Todd Austin, Dennis Sylvester, et al. Energy-efficient subthreshold processor design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(8):1127–1137, 2009. 1.5

[182] Eddy Z Zhang, Yunlian Jiang, and Xipeng Shen. Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? In *ACM Sigplan Notices*, volume 45, pages 203–212. ACM, 2010. 5.3.4.1

[183] Lei Zhang, Mei Yang, Yingtao Jiang, Emma Regentova, and Enyue Lu. Generalized wavelength routed optical micronetwork in network-on-chip. In *Proceedings of the 18th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 698–703, 2006. 6.1.1

[184] Michael Zhang and Krste Asanovic. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In *Proceedings of the 32nd annual International Symposium on Computer Architecture (ISCA)*, pages 336–345, 2005. 7

[185] Hongzhou Zhao, Arrvindh Shriraman, and Sandhya Dwarkadas. SPACE: Sharing Pattern-based Directory Coherence for Multicore Scalability. In *19th international conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 135–146, 2010. 5.4

[186] Hongzhou Zhao, Arrvindh Shriraman, Sandhya Dwarkadas, and Vijayalakshmi Srinivasan. SPATL: Honey, I Shrunk the Coherence Directory. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 33–44, 2011. 5.1, 5.4

[187] Li Zhao, Ravi Iyer, Srihari Makineni, Don Newell, and Liqun Cheng. Ncid: a non-inclusive cache, inclusive directory architecture for flexible and efficient cache hierarchies. In *Proceedings of the 7th ACM international conference on Computing frontiers*, pages 121–130, 2010. 3.1.1

[188] Xuezhe Zheng, Dinesh Patil, Jon Lexau, Frankie Liu, Guoliang Li, Hiren Thacker, Ying Luo, Ivan Shubin, Jieda Li, Jin Yao, Po Dong, Dazeng Feng, Mehdi Asghari, Thierry Pinguet, Attila Mekis, Philip Amberg, Michael Dayringer, Jon Gainsley, Hesam Fathi Moghadam, Elad Alon, Kannan Raj, Ron Ho, John E. Cunningham, and Ashok V. Krishnamoorthy. Ultra-efficient 10gb/s hybrid integrated silicon photonic transmitter and receiver. *Optics Express*, 19(6):5172–5186, 2011. 6.3.1.1