UNIVERSIDAD DE MURCIA

Facultad de Informática

Sincronización y Comunicación Eficientes en Arquitecturas Many-Core CMP

Efficient Synchronization and Communication in Many-Core Chip Multiprocessors

José Luis Abellán Miguel

2012

UNIVERSIDAD DE MURCIA

Facultad de Informática

Departamento de Ingeniería y
Tecnología de Computadores

Sincronización y Comunicación Eficientes en Arquitecturas Many-Core CMP

TESIS DOCTORAL

Autor:
José Luis Abellán Miguel

Directores:
Juan Fernández Peinador
Manuel Eugenio Acacio Sánchez

Murcia, Septiembre de 2012

Resumen

En la última mitad del siglo XX los computadores sufrieron una constante evolución en rendimiento propiciada principalmente por el aumento de la escala de integración, que habilitaba cada vez más transistores disponibles para incorporar novedades arquitectónicas, o para aumentar el tamaño de las memorias en el chip. Esta tendencia fue ya predicha por Gordon E. Moore [42], quien ya en 1965 expuso que el número de transistores por área de silicio sería doblado cada dieciocho meses por los sucesivos avances tecnológicos. La otra causa importante para el aumento del rendimiento fue el crecimiento incesante de la frecuencia de procesamiento en cada nueva generación de chips, haciendo la circuitería cada vez más rápida. De este modo, los microprocesadores alcanzaron niveles de complejidad demasiado altos requiriendo de procesos de verificación cada vez más costosos en tiempo y valor económico. Disponiendo de un sólo núcleo de procesamiento, estas arquitecturas uniprocesador estaban fundamentalmente enfocadas a la extracción de paralelismo a nivel de instrucción (*Instruction-Level Parallelism* en inglés, o ILP), en el que se aprovechaba al máximo la ejecución simultánea de instrucciones que podían emitirse de manera independiente. Los esfuerzos de los arquitectos de computadores para extraer cada vez más ILP, originaron nuevos diseños que incluían ejecución fuera de orden y especulativa, predictores de saltos muy sofisticados, cauces de ejecución segmentados con muchas etapas y con una anchura de emisión considerable, etc. Aún así, el beneficio de rendimiento obtenido se veía cada vez más limitado por la influencia de: las dependencias verdaderas entre instrucciones por los riesgos de datos tipo *Read-After-Write* (RAW); los riesgos de control por las instrucciones de salto; o por los llamados riesgos estructurales, que serializan la ejecución de instrucciones que presentan un conflicto en el uso de un mismo recurso hardware como una unidad funcional aritmético-lógica.

A partir del presente siglo, tanta complejidad se unió al hecho de que la frecuencia de operación estaba alcanzando niveles en los que la temperatura de los chips se estaba volviendo inmanejable. Por ello, se inició un proceso de renovación de los procesadores en los que se redujo la frecuencia de operación, pero se incorporaron varios núcleos de procesamiento en el mismo chip. Esto condujo a un nuevo concepto de arquitectura denominada multiprocesador en un sólo chip o CMP. En los diseños de los CMPs, se primó más la extracción del paralelismo a nivel de hilo (*Thread-Level Parallelism*, o TLP) que intentar extraer más ILP. Así, el rendimiento por ciclo de ejecución comenzó a aumentar de nuevo consiguiéndose superar a los mejores uniprocesadores desarrollados hasta la fecha. A medida que más y más núcleos iban integrándose en los CMPs, estos sistemas evolucionaron a lo que hoy en día se denomina CMPs de muchos núcleos, o *many-core* CMPs. Los *many-core* CMPs están alcanzando ya el centenar de núcleos de ejecución como el procesador Tile-Gx con 100 núcleos [145]. Para simplificar el diseño de estas arquitecturas y para hacerlas escalables, se diseñan en base a un esquema modular, en el que se define un bloque básico de cómputo compuesto por un núcleo, niveles de caché privados y un fragmento de una caché compartida global, además de un enrutador que conecta con el resto de bloques de cómputo. La manera en la que se conectan todos los bloques que conforman un *many-core* CMP está basado en una red escalable punto-a-punto como una malla 2D. Otro aspecto importante de estos sistemas es que su programación tiene que ser lo más sencilla posible para simplificar la tarea al programador. Lo más común es que estos sistemas implementen un sistema de memoria compartida [29]. Mediante éste, las operaciones de comunicación y sincronización entre los hilos de ejecución se realiza mediante operaciones de acceso a posiciones de memoria convencionales, así como instrucciones especiales para el caso de la sincronización como *Load-Linked/Store-Conditional* (*LL/SC*), o de lectura-modificación-escritura como *test&set*.

Sin embargo, desde el año 2004 se ha experimentado un proceso de disminución en el crecimiento constante del rendimiento principalmente por motivos de consumo excesivo de energía, tamaño de los transistores alcanzando la escala atómica, grandes costos de diseño, fabricación y verificación de los chips, etc. Lo cual ha hecho que de nuevo se reconsideren las arquitecturas actuales para idear una manera de mejorarlas sin tales problemas iniciándose una tendencia denominada *More-than-Moore* [150]. Mediante esta, los nuevos diseños arquitectónicos comparten circuitería basada en tecnología digital CMOS y tecnología menos escalable pero más rápida en transmisión de señales como las no digitales: analógica, RF u óptica.

En esta tesis hemos identificado tres de los mayores cuellos de botella para el rendimiento y escalabilidad de las arquitecturas *many-core* CMP. En particular, los mecanismos de sincronización de barrera y cerrojo cuando presentan alta contención, es decir, cuando hay un gran número de hilos competiendo por el uso de la barrera o bien por el acceso a la sección crítica (SC) que protege el cerrojo. Téngase en cuenta que habrá más probabilidad de contención conforme más TLP se esté extrayendo de arquitecturas *many-core* CMPs con cada vez más núcleos. Otro problema identificado es la eficiencia en el mantenimiento de la coherencia del uso de los bloques de memoria en todos los niveles de la jerarquía de memoria de estos sistemas de memoria compartida. Un protocolo de coherencia implementado en hardware para mayor eficiencia será el encargado de llevar a cabo esta tarea. A medida que el número de núcleos sea cada vez mayor en las arquitecturas *many-core* CMP, mayor será la actividad del protocolo para garantizar la coherencia de todas las posiciones de memoria compartidas que se utilicen para comunicar y sincronizar cada nueva generación de *many-core* CMPs con un mayor número de hilos. Para paliar estas deficiencias en el rendimiento y aprovechar más el rendimiento de estas arquitecturas, hemos propuesto tres mecanismos hardware que se explicarán más abajo: *GBarrier*, para un mecanismo de barreras eficiente; *GLock*, que proporciona un manejo de la contención en el acceso a las SC protegidas por cerrojos de manera justa y eficiente; y *ECONO*, un protocolo de coherencia muy simple que aporta gran eficiencia a bajo costo. Además, siguiendo la tendencia *More-than-Moore*, hemos considerando tecnología analógica actual llamada *G-Lines* para obtener un mayor rendimiento en nuestras propuestas. *G-Lines* utilizan tecnología analógica para transmitir señales de un sólo bit a lo largo de una línea que puede abarcar la totalidad de un chip a muy bajo costo y en un sólo ciclo de reloj. Por otro lado, también hemos considerado una implementación de las mismas basadas en una metodología estándar de diseño actual que se utiliza en los procedimientos convencionales de fabricación de chips, a la que llamaremos simplemente *Estándar*.

A continuación se exponen nuestras tres propuestas junto a los resultados más relevantes que hemos obtenido en los estudios realizados en esta tesis. Para superar el cuello de botella que supone la ejecución de barrera en máquinas *many-core* CMP de memoria compartida, hemos propuesto *GBarrier*. *GBarrier* es un novedoso mecanismo hardware especialmente diseñado para la ejecución de barreras de manera eficiente eliminando todas las limitaciones de rendimiento en las implementaciones basadas en software, e incluso en otras muchas, también hardware, implementadas hasta el día de hoy. En particular, nuestra propuesta consiste en dos componentes principales. El primero es una red especializada

de muy bajo costo integrada en el CMP que puede ser escalada fácilmente utilizando una esquema jerárquico. El segundo componente es un protocolo de sincronización muy simple que es ejecutado por los controladores de la red especializada. Las razón principal por la cual *GBarrier* es mucho más eficiente que la inmensa mayoría de las propuestas actuales, basadas en el uso de operaciones sobre posiciones de memoria compartida para la sincronización, se debe a que para la sincronización nuestro mecanismo no afecta al sistema de memoria en absoluto. La gran ventaja de esto es que evitamos toda la actividad de coherencia y todo el tráfico de red debido a la operación de barrera que las otras implementaciones necesitan y que restringe la escalabilidad, como explicamos anteriormente. Nuestra propuesta ha sido escenificada en otros contextos de operación exponiendo cómo podríamos abordarlos satisfactoriamente: la sobrecarga en la integración de varias *GBarriers* en un mismo CMP; cuando no todos los hilos de ejecución disponibles utilizan la barrera (*GBarrier* para subconjuntos de todos los núcleos del CMP); y el uso de *GBarrier* en arquitecturas con núcleos con soporte para múltiples hilos simultáneos (arquitecturas SMT).

Para evaluar *GBarrier*, hemos considerado dos implementaciones para sintetizar nuestra infraestructura de barrera utilizando la tecnología de *G-Lines* y la *Estándar* anteriormente mencionadas. Un estudio preliminar de rendimiento potencial de ambas revela que las diferencias en términos de consumo de área y de energía pueden considerarse despreciables por los mínimos recursos hardware que requiere nuestro diseño (unos pocos controladores y cableado de anchura de un sólo bit). La diferencia estriba en que la tecnología de *G-Lines* proporciona mayor rapidez en las transmisiones de nuestro protocolo de sincronización, consiguiendo una barrera de menor latencia. Por otro lado con la *Estándar*, nuestro mecanismo se podría plasmar directamente en un circuito siguiendo los procesos actuales de fabricación de chips. Hemos integrado ambas implementaciones para *GBarrier* en un entorno de simulación denominado Sim-PowerCMP. A partir del cual, hemos modelado un *many-core* CMP de 32 núcleos de ejecución sobre el que ejecutar una serie de aplicaciones paralelas para obtener el rendimiento que ofrece nuestra propuesta (*benchmarks*). Así, hemos comparado rendimiento utilizando la mejor barrera software hasta el momento (*barrera en árbol*), tres núcleos de aplicación (o *kernels*) y tres aplicaciones científicas. Como conclusión de este estudio podemos decir que en cuanto a tiempo de ejecución ambas implementaciones obtienen reducciones en tiempo de ejecución muy similares, por tanto nuestro diseño para barrera no depende de tecnología no estándar para obtener una implementación de barrera eficiente en *many-core* CMPs. Más específicamente, para los *kernels* y aplicaciones científicas, las reducciones medias

en tiempo de ejecución son del 54 % y del 21 %, del 53 % y del 18 % en tráfico de red. La razón es porque nuestra propuesta evita el tiempo dedicado al mantenimiento de coherencia de las posiciones de memoria compartidas que la implementación software emplea, y tampoco inyecta tráfico de red en la red de interconexión principal del CMP, evitando así congestión de la red por este tráfico que compite por los mismos recursos de red que el tráfico de petición y recepción de datos de las aplicaciones paralelas ejecutadas. Por otro lado, hemos evaluado la reducción del consumo de energía utilizando para ello la métrica *energy-delay*^{2} *product* (ED^{2}P), obteniendo reducciones medias del 76 % y 31 % para los kernels y aplicaciones científicas respectivamente.

Por otro lado, para superar los problemas de rendimiento y escalabilidad de las operaciones de sincronización mediante cerrojo, hemos propuesto *GLock*. Nuestra propuesta está especialmente diseñada para cuando existe una alta contención en el acceso a las SC protegidas por el cerrojo, porque en este caso, las implementaciones tradicionales de cerrojo tiene serios límites de rendimiento y escalabilidad. *GLock* está basado en dos componentes principales. El primero es una red especializada de muy bajo costo integrada en el CMP, mientras que el segundo es un protocolo de sincronización basado en paso de mensajes y en transferencia de un token que será el que determine el hilo que tiene el acceso exclusivo a la SC. Debido al hecho de que nuestra propuesta se dedica a los cerrojos altamente contendidos, su ejecución se puede simultanear con otras implementaciones de cerrojos que no presentan una alta contención, tales como las implementaciones tradicionales software que utilizan instrucciones especializadas *test-and-test&set*. Como consecuencia de un análisis que hemos llevado acabo sobre el conjunto de benchmarks que hemos utilizado para evaluar nuestra propuesta, hemos obtenido que el número de cerrojos distintos que están altamente contendidos es bastante reducido. En particular, hemos detectado un máximo de dos cerrojos en nuestros benchmarks con alta contención por lo que hemos integrado dos *GLocks* para la evaluación.

La evaluación de *GLock* también se ha llevado a cabo utilizando los dos tipos de tecnología *G-Lines* y *Estándar*, llegando a las mismas conclusiones que para *GBarrier*, es decir, existen diferencias poco significativas en el rendimiento que ofrecen ambas propuestas, con lo que *GLock* no depende de tecnología no estándar para obtener una implementación muy eficiente para cerrojos altamente contendidos; la tecnología de *G-Lines* proporciona la implementación más rápida; y finalmente, la *Estándar* reduce los costes derivados de la fabricación del *GLock* en un chip, por seguir una metodología estándar de desarrollo. Ambas implementaciones han sido integradas en Sim-PowerCMP para un *many-core*

CMP de 32 núcleos, comparando el rendimiento de nuestra propuesta frente a la mejor implementación software hasta la fecha (MCS [77]), y hemos utilizado un conjunto significativo de benchmarks basados en microbenchmarks y aplicaciones científicas. Las estadísticas de rendimiento ofrecen las siguientes reducciones medias frente a MCS: en el tiempo de ejecución del 42 % y 14 % para los microbenchmarks y aplicaciones científicas respectivamente; en tráfico, del 76 % y 23 %; y en consumo de energía utilizando la métrica ED^2P, del 78 % y 28 %, para los microbenchmarks y aplicaciones científicas respectivamente.

Finalmente, se ha propuesto también en esta tesis un protocolo de coherencia para *many-core* CMPs denominado *Express Coherence Notification*, o simplemente *ECONO*. En el mantenimiento de la coherencia, *ECONO* hace uso de mensajes especiales que son enviados de manera atómica y en difusión (*broadcast*) llamados mensajes *Atomic Coherence Notification* (*ACN*). Estos mensajes son enviados a través de una red dedicada de mínimo coste implementada mediante tecnología de *G-Lines* para máxima eficiencia (experimentos demuestran que la tecnología *Estándar* no es capaz de obtener gran rendimiento para envíos de mensajes en *broadcast* para todos los núcleos de una arquitectura *many-core* CMP). Para la implementación de *ECONO*, hemos obtenido una primera versión, *4-hop ECONO*, que utiliza cuatro saltos en el camino crítico para hacer coherente una escritura sobre un bloque que produce un fallo de caché, y que requiere actuación del protocolo sobre otras copias del bloque de datos. Esta versión ha sido mejorada introduciendo el concepto de mensajes *ACN* imprecisos que envían en *broadcast* un subconjunto de la dirección del bloque en cuestión, y también hemos reducido a tres saltos el camino crítico obteniendo la versión *3-hop ECONO*, reduciendo así el tráfico por la red de interconexión del CMP y, por tanto, el consumo de energía.

Para evaluar las distintas implementaciones de *ECONO*, hemos simulado un *many-core* CMP de 16 núcleos usando Simics-GEMS. En cuanto al máximo rendimiento potencial, hemos obtenido que *ECONO* con una red especializada compuesta por tres líneas globales implementados mediante tecnología de *G-Lines*, ofrece el mejor compromiso entre latencia del mensaje y costo en área. Por otro lado, en cuanto a la implementación de *ECONO* con mensajes *ACN* imprecisos, la configuración *Index+5* es la mejor opción. Finalmente, hemos cuantificado los beneficios en rendimiento de las anteriores versiones de *ECONO* en comparación a dos protocolos de coherencia contemporáneos: *Hammer* y *Directory*. Las principales conclusiones de este estudio son las siguientes: nuestra propuesta tiene el diseño más simple, requiere una sobrecarga en área del chip similar a *Hammer*, obtiene un rendimiento similar a *Directory*, y constituye la

implementación más eficiente en términos de energía. Finalmente, dada la gran simplicidad de nuestra propuesta, hemos identificado una serie de optimizaciones que se podrían aplicar y que utilizan algunos protocolos actuales para mejorar el rendimiento, tales como filtrado de mensajes para reducir el costo energético del envío atómico y en *broadcast* de los *ACNs*, o bien utilizar el concepto de coherencia directa, en el que se evita dirigir mensajes al directorio del protocolo, reduciendo el número de saltos en el camino crítico en el mantenimiento de la coherencia enviando dichos mensajes directamente el poseedor del bloque de datos solicitado. Podríamos implementar esto último incluyendo en el mensaje *ACN*, que es recibido de manera global, el identificador del siguiente propietario del bloque.

De todo lo anterior podemos asegurar que nuestras propuestas resuelven de una forma eficiente los problemas de rendimiento derivados de implementaciones ineficientes para sincronización mediante barrera y cerrojo en situaciones de alta contención, y de los protocolos de coherencia que han de gestionar un número mayor de compartidores de bloque, en las arquitecturas *many-core* CMP.

Índice

A mi esposa, padres y hermanos

Agradecimientos

La elaboración de esta tesis ha sido un gran reto que no habría podido superar sin la ayuda de aquellas personas que forman parte de mi vida. A ti Inma, mi esposa, por todo lo que significas para mí, lo que me has ayudado en todos estos años, lo que hemos sufrido y disfrutado juntos, por tus consejos, el ánimo que siempre me das, tu cariño y mucho más. A mis padres por la educación y todos los valores que he recibido de ellos que me han ayudado a abrir puertas en la vida, por el ánimo que siempre me habéis dado, por vuestra ayuda incondicional y por vuestros consejos. Mamá, porque eres un ejemplo de superación para mí, por tu constancia y perfeccionismo. Papá, por el ejemplo de responsabilidad y lucha que me ha ayudado a ser más fuerte, tu disciplina y organización. A mis hermanos que siempre están ahí y a los que intento dar humildemente ejemplo. Alicia, por tus consejos, tu ayuda y por creer en mí. Pedro, aunque eres aún pequeño, paso muy buenos momentos contigo que me ayudan a desconectar. A mis suegros, Antonia y Joaquín, que me han ayudado en todo y me han acogido como a un hijo desde el primer día. También a mis cuñados, Alex, Marian y Mario, con los que me divierto mucho y con los que es un placer estar. A mis primos-hermanos Alberto y Juan Antonio, con los que he compartido grandes momentos a lo largo de mi vida, con los que he disfrutado y reído mucho. A mis tíos y abuelos, sólo espero que estéis orgullosos de mí y que sepáis que os he echado de menos. También me gustaría dar las gracias a familiares no tan directos como Justo Ireno, tu experiencia como Ingeniero Informático siempre me ha servido de timón de barco, y también a Antonio, por nuestras charlas sobre informática y tecnología que realmente son un gozo.

Mencionar también a mis compañeros de fatiga durante los largos años de carrera como mi amigo Miguel Ángel y mi amigo Fernando, éste último con el que también he convivido en el máster. Sin vosotros desde luego que hubiera sido muchísimo más difícil alcanzar los objetivos. Como no, también nombrar

a mis compañeros de trabajo y amigos: Alberto, Ana, Antonio, Chema, Dani, Dani Sánchez, Epi, Ginés, Juanma, Ricardo, Rubén y Toni. Gracias por hacer del laboratorio un lugar muy agradable de trabajo y por la gran ayuda que me habéis ofrecido en todas las ocasiones que os he requerido.

Thanks to professor Davide Bertozzi for giving me the opportunity to work as a part of his team at UNIFE. My stay in Ferrara was most definitely an enriching experience where I learnt a lot, and I worked with very hard-working and wonderful guys, willing always to help me, such as Daniele Ludovici, Hervè Tatenguem and Alessandro Strano. They really allowed me to spend a very pleasant internship that I will never forget. I would like also to thank my colleague from Bologna, Andrea Marongiou, for his kindness and all those hours we spent discussing designs and results.

A mis directores de tesis, Manolo y Juan, los mejores directores de tesis que podría tener. Vuestra profesionalidad, disciplina, constancia, entrega y pasión por vuestro trabajo, me han ayudado a madurar muchísimo y a ser lo que soy a día de hoy. Gracias por darme la oportunidad de iniciar mis estudios de doctorado en el departamento, por creer en mí desde un primer momento e intentar siempre orientarme por el mejor camino. Sin duda, me habéis hecho disfrutar de este mundo apasionante de la investigación.

Finalmente, agradecer también a la Fundación Séneca (Agencia de Ciencia y Tecnología de la Región de Murcia) por su vital apoyo económico mediante una Beca-Contrato Predoctoral de Formación del Personal Investigador (FPI).

A todos, ¡MUCHAS GRACIAS!

Universidad de Murcia

Facultad de Informática

Departamento de Ingeniería y
Tecnología de Computadores

Efficient Synchronization and Communication in Many-Core Chip Multiprocessors

PhD Thesis

By
José Luis Abellán Miguel

Advised by
Juan Fernández Peinador
Manuel Eugenio Acacio Sánchez

Murcia, September 2012

Abstract

Multicore architectures (chip-multiprocessors or CMPs) constitute nowadays the best way to take advantage of the increasing number of transistors available in a single die. In particular, they provide higher performance and lower energy consumption than more complex unicore architectures. This is due to the fact that these architectures mainly focus on exploiting thread-level parallelism (TLP) rather than instruction-level parallelism (ILP). As the number of cores increases in these throughput-oriented machines, they are referred to as many-core CMPs. To ease the programmability task, these systems commonly adopt a shared-memory programming model in which communications and synchronizations among threads are accomplished by means of memory access instructions on shared variables. This thesis focuses on outperforming three of the major problems that restrict efficiency and scalability in future shared-memory tiled many-core CMPs: the synchronization operations of barriers and locks, and the cache coherence protocol.

Regarding barrier synchronization, traditional software-based barrier implementations for shared memory parallel machines tend to produce hot-spots in terms of memory and network contention as the number of cores increases. To completely remove such negative side effects we develop *GBarrier*. Our proposal is a hardware-based barrier mechanism especially aimed at providing efficient barriers in future many-core CMPs. To this end, our proposal deploys a dedicated *G-Line*-based network to allow for fast and efficient signaling of barrier arrival and departure. Since *GBarrier* does not have any influence on the memory system, we avoid all coherence activity and barrier-related network traffic that traditional approaches introduce and that restrict scalability. To implement *GBarrier*, we consider two different technologies. The first is a state-of-the-art full-custom technology, namely *G-Lines*, whilst the second is a cost-effective mainstream industrial toolflow with an advance 45 nm technology, or *Standard* technology for

short. Both *GBarrier* implementations report very similar reductions in execution time, thus not making our proposal so dependent on a full-custom technology to achieve extremely efficient synchronization in many-core CMPs. In particular, through detailed simulations of a 32-core CMP, we compare *GBarrier* against one of the most efficient software-based barrier implementations for a set of kernels and scientific applications. Evaluation results bring important average reductions when the kernels and scientific applications are considered: 54% and 21% in execution time, respectively; 53% and 18% in network traffic, respectively; and also 76% and 31% in the energy-delay^2 product metric for the full CMP, respectively.

With respect to lock synchronization, this is a key programming primitive for shared-memory many-core CMPs. The problem is that, as the number of cores increases so does the degree of contention that a single lock may exhibit, but in this case, conventional software implementations cannot meet the desirable levels of performance and scalability. Meanwhile, most existing hardware-supported lock proposals require modifications at some level of the memory hierarchy, thus degrading QoS of applications through synchronization traffic. To overcome such performance limitations, we propose *GLock*, a dedicated network infrastructure along with a token-based message-passing protocol to provide a non-intrusive, extremely efficient and fair implementation for highly-contended locks. As for *GBarrier*, two implementations of *GLock* that leverage *G-Lines* and *Standard* technologies are considered. We conclude that by leveraging both technologies significant reductions in execution time can be obtained with a negligible performance gap between them. Besides, both implementations require a minimal power dissipation and a marginal on-chip area overhead. As a result, our *GLock* proposal is not so dependent on full-custom technology to provide very efficient synchronization for highly-contended locks in many-core CMPs. More specifically, when compared *GLock* against the most efficient software-based lock using a set of microbenchmarks and real applications, we obtain average reductions of: 42% and 14% in execution time, respectively; 76% and 23% in network traffic, respectively; and 78% and 28% in the energy-delay^2 product (ED^{2}P) metric for the full CMP, respectively.

Finally as to the coherence protocol, the design of an efficient coherence protocol for shared-memory many-core CMPs should take into account several aspects related to efficiency such as on-chip area overhead, energy consumption, and performance. Nevertheless, another important metric to be considered is its resulting complexity. In fact, one may opt to sacrifice some efficiency in exchange for a simpler verification process. The problem is that all these requirements are

very difficult to meet in a single coherence protocol at once. To accomplish this, we propose *Express Coherence Notification* (*ECONO*), a cache coherence protocol aimed at providing simultaneously a simple and efficient design. To maintain coherence, *ECONO* relies on express coherence notifications which are broadcast atomically over a dedicated lightweight on-chip network. For the implementation of this special network, differently to *GBarrier* and *GLock*, we only rely on the state-of-the-art *G-Lines* technology since the *Standard* technology is not enough to obtain an efficient and very fast implementation for the *ECONO*'s notifications. We implement and evaluate *ECONO* utilizing full-system simulation and a representative set of benchmarks. As compared to two contemporary coherence protocols, *Hammer* and *Directory*, *ECONO* has the simplest design, requires an area overhead similar to *Hammer* and reports performance results similar to *Directory*. Particularly, *ECONO* achieves an average reduction of 2% in execution time and an average reduction of 3% in network traffic, when compared to *Directory*. In addition, our experimental evaluation also leads to conclude that *ECONO* is also the most energy efficient design.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For the last half-century, computers have been doubling in performance and capacity every couple of years. Such phenomenal progress is a consequence of the well-known Moore's Law that has been basically sustained till 2000s thanks to the continuous advances in semiconductor technology, to obtaining ever-shrinking transistor sizes, and achieving ever higher clock frequencies. This allowed computer architects to make use of the ever-increasing amount of silicon resources to design more and more sophisticated and ever-faster pipelines in uniprocessor systems, in order to better exploit the instruction level parallelism (ILP) present in sequential programs. Nonetheless, at the beginning of this century this successful strategy has come to an end mainly due to the thermal-power issues. The response was a shift towards parallel architectures that mainly focus on exploiting thread level parallelism (TLP) rather than ILP, laying the foundation of the core era. Multicore architectures, multiprocessor in general, are systems specially tailored to the exploitation of massive throughput by incorporating many simpler and lower-frequency computing units. This paradigm shift towards this throughput-oriented machines brings about new fundamental challenges to harness their ever-increasing peak potential power. Therefore, it is imperative that programmers can deal with a simple and efficient programming model such as a shared-memory programming model [29]. Nevertheless, computer architects must struggle to mitigate some performance bottlenecks related to this intuitive programming model. For instance, when considering synchronization operations, such as locks and barriers, or when ensuring coherence across all levels of a memory hierarchy through a cache coherence protocol implemented in hardware.

1.1 Towards Many-core CMPs

The steady evolution of computing has been made possible mainly thanks to the contribution of the three following factors: *technology scaling* that propitiates opportunities for *architectural innovations* and *advances in compilation*. The technology roadmap for semiconductor technology was already predicted by Gordon E. Moore who stated in 1965 that the number of transistors per silicon area would double every eighteen months due to the transistors getting smaller every successive process technology [42]. This is commonly known as the Moore's Law and, surprisingly, it is expected to remain valid and continue well into the future [5,16].

Over the years, we can identify two major different trends that have marked and guided manufacturing of every new computing architecture at industry. The first-stage trend that took a longer period of time was valid till approximately year 2000 by following a virtuous cycle, by which computer architects obtained higher performance in every new design basically by shrinking feature sizes and increasing frequency of circuits. In this way, an increased transistor density enabled more and more space on chip for incorporating ever complex designs with very-deep instruction pipelines [43], highly speculative [117], out-of-order processors [46] and larger on-chip cache hierarchies [109]. The main purpose to do so was to increase the amount of work performed in each cycle allowing more and more capability to execute multiple instructions from the same (sequential) program simultaneously, i.e., the extraction of ILP. However, since year 2000 and despite new progress in integration technology, the efforts to design very aggressive and very complex wide issue superscalar processors in this trend came to a stop. While there is still a little ILP left to be extracted, RAW (read-after-write) data hazards between consecutive instructions, control hazards due to branch instructions, as well as structural hazards which serialize the execution of instructions that try to use the same hardware resource at the same time (i.e. an arithmetic and logic unit, or ALU), made impossible to increase performance without a considerable effort [72]. Moreover, even the investment of the ever larger on-chip caches reached the point of diminishing return [86]. As a result, every new performance improvement has been empirically close to the square root of the number of required transistors [72]. Besides, a higher and higher clock frequency for a faster circuitry involves important heat problems and high energy consumption. Moreover, the efforts to maintain manageable parameters for the thermal-power issues such as increased threshold voltage to control leakage, or limited supply-voltage scaling, decrease the performance benefits of transistor

scaling [56]. All above problems were identified as the *Power*, *Memory*, and *ILP Walls* [85].

The second-stage and current trend consists of a paradigm shift towards multi-core architectures. Instead of scaling performance by improving single core performance as was done in the past, the growing of performance is achieved by putting multiple cores onto a single chip and usually connecting them through a shared memory [30], thereby effectively integrating a complete multiprocessor on one chip. This type of systems are commonly referred to as chip multiprocessors or CMPs. In this systems, rather than ILP, they improve system performance by exploiting thread-level parallelism (TLP). CMPs have important advantages over very wide-issue out-of-order superscalar processors. In particular, they provide higher aggregate computational power, multiple clock domains, better power efficiency, and simpler designs. Additionally, the use of simpler cores reduces the cost of design and verification in terms of time and money.

While the number of cores currently offered in general-purpose CMPs has already gone above ten (e.g. the 6-core 2-die AMD's Magny-Cours design [115], the 18-core BlueGene/Q [114], or the 16-core SPARC T3 [73]), now that the Moore's Law will make it possible to double the number of processing cores per chip every 18 months [86], very soon there will be available on-chip the resources required to integrate dozens of cores or even hundreds of them. CMPs of this kind are commonly referred to as many-core CMPs. Examples of this new generation of CMPs can be the following. The 48-core Single-chip Cloud Computer [55], an experimental research microprocessor developed by Intel in the context of the Tera-scale Computing Research Program. The Intel Polaris [92] which is a prototype with 80 cores. And the Tile-Gx Processor [145] from Tilera Corporation which is comprised of up to 100 cores on a chip. For the success of this kind of systems in the future, all elements that could compromise system scalability must be avoided. One of such elements is the interconnection network. As stated in [124], the area required by a shared interconnect, like a bus, as the number of cores grows has to be increased to the point of becoming impractical. Hence, it is necessary to turn to a scalable interconnection network.

In this thesis, we focus on tiled many-core CMPs [93, 103] which are designed as arrays of identical or close-to-identical building blocks known as tiles, that provide a scalable alternative to current small-scale CMP designs and help in keeping complexity manageable. In these architectures, each tile is comprised of a processing core (or even several cores), one or several levels of caches, and a network interface or router that connects all tiles through a tightly integrated and lightweight point-to-point interconnection network (e.g., a two-dimensional

mesh). Differently from shared networks, point-to-point interconnects are suitable for many-core CMPs because their peak bandwidth and area overhead scale with the number of cores.

1.2 A Shared-Memory Programming Model

The different threads of a parallel program need to communicate and synchronize in order to carry out a task cooperatively to completion. For this matter, two popular types of general-purpose communication abstractions exist, which provide a link between the software (programming model) and the hardware (physical implementation). Threads can exchange information by sending messages (message passing model), or by merely accessing and modifying shared memory locations (shared memory model).

This thesis focuses on a shared memory model, as is widely regarded as a more intuitive model than message passing for the development of parallel programs [90] and nowadays is the prevalent model in most CMPs [72], and the common belief is that future many-core CMP architectures will also implement this memory model. Particularly, the hardware-managed, implicitly-addressed, coherent caches memory model [75]. With this memory model, all on-chip storage is used for private and shared caches that are kept coherent by hardware. Communication between threads is performed by writing to and reading from shared memory. In order to guarantee the integrity of shared data structures, most current systems support synchronization through a combination of hardware (special instructions, such as *LL/SC*, or atomic read-modify-write instructions, such as *test&set*, that operate on shared memory) and software (higher-level mechanisms such as locks or barriers implemented atop the underlying hardware primitives) [29].

1.3 The Slowdown in Exponential Growth of Performance

In the near term, there are new challenges to tackle with in the multicore revolution. It is due to the fact that, although for every technology generation the transistor integration is doubled and the number of cores on chip grows apace, a flattened curve for the growth of performance has been appreciated [127] mainly due to reducing power and frequency parameters to alleviate the thermal-power

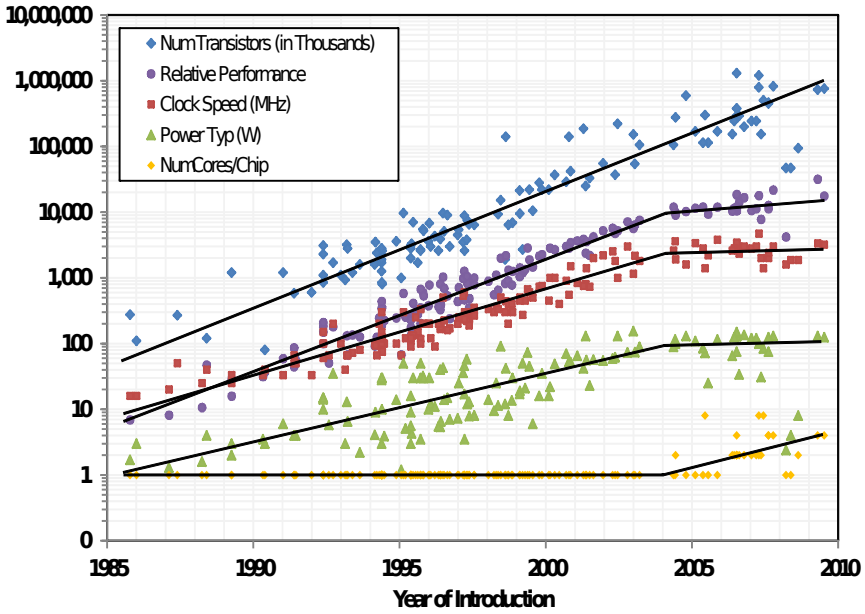


Figure 1.1: Transistors, frequency, power, performance, and cores over time (1985-2010). *Source: D. Patterson, UC-Berkeley.*

issues by adjusting to a realistic power budget and thermal envelope. Figure 1.1 depicts such a new trend observed since 2004 that shows flattening curves for performance, frequency and power, while the number of transistors and cores keep the expected growing. This demands new research efforts to develop new architectural solutions in order to continue with the exponential growth of performance to be kept alive in the future. In an attempt to continue in that direction, this thesis identifies two of the most severe performance bottlenecks in many-core CMPs, synchronization and the maintenance of cache coherence, and proposes hardware-based infrastructures to mitigate them as the core count increases in future developments. To gain insight into these main problems to continue with scalable performance, the next two sections delve into them and introduce our proposals.

1.4 The Synchronization Problem

By relying on a shared-memory programming model to execute parallel applications in many-core CMPs, conventional implementations of synchronization operations, such as barrier and locks, rely on shared variables which are atomically

updated. In particular, when considering global barriers and highly-contended locks (i.e. there is a significant amount of threads requesting the lock at the same time), without the proper hardware support, this kind of software-based implementations cannot provide good scalability as the number of cores increases.

Regarding barrier implementations in software, as we will discuss in Chapter 3, the use of shared variables implies a performance bottleneck, an ever-growing amount of resources and high energy consumption requirements. In more depth, the cache coherence protocol must come into play to maintain memory consistency across all levels of the memory hierarchy. In turn, coherence activity translates into traffic injection in the interconnection network that may interfere with application-related traffic. On the other hand, the busy-waiting required to wait for the completion of the barrier synchronization on locally-cached shared variables has also significant implications on the energy consumed by the L1 caches.

As to the software implementations for the highly-contended locks, as we will expose in Chapter 4, these operations are very critical to performance since lock contention causes serialization. Therefore, an implementation based on the use of shared variables is not efficient enough due to the performance bottlenecks, the ever-growing amount of resources and high energy consumption requirements, that they produce as explained above.

In this thesis we address these two problems separately. Particularly, we deal with neither shared variables nor traffic injection into the main interconnect to implement these mechanisms for tiled many-core CMPs. Instead, we design a dedicated on-chip network infrastructure to implement a very efficient hardware-based barrier, and another one that achieves very efficient hardware support for highly-contented locks.

1.5 The Cache Coherence Problem

The communication and synchronization operations among threads in shared-memory parallel machines occurs implicitly as a result of reading from and writing to shared variables. The order by which all threads see the changes in the shared variables is defined according to a particular memory consistency model [133]. CMPs normally include a memory hierarchy with one or more levels of private caches to each core to avoid the increasing gap between processor and memory speeds (i.e. the *Memory Wall* [149]). The reason is that the smaller and faster private levels absorb the vast majority of memory accesses due to the

exploitation of temporal and spatial locality that applications exhibit, thereby reducing the average memory access latency and network traffic in the CMP's interconnect. As a result, in a given instant of time, there could be several copies of a particular memory location (or block) in the private caches. If a core modifies its local copy without any further action, subsequent memory accesses on the remaining sharers' copies of the memory block would read or write a stale block resulting in data incoherence. While this task could be solved in software by programmers (like in message-passing programming) a hardware-based coherence protocol is commonly in charge to do that. Clearly, the main benefit is a simplification in programming because all caches are completely transparent to software. Moreover, the coherence protocol precludes application's execution from any data incoherence by guaranteeing that each read to every memory block returns the latest value written to it, and the semantic is that each write to the same memory location appears to be seen in the same order by all processors [87].

From all above, we can affirm that the efficiency of communication and synchronization operations among threads is highly dependent upon the efficiency of the cache coherence protocol. In consequence, a great deal of attention has long been devoted to the development of cache coherence protocols, with a first-order goal of achieving scalability and efficiency in each new generation of shared-memory parallel machines. In this way, in the late 80s and the beginning of the 90s there were a number of shared memory multiprocessors with a processor count even reaching several hundreds in a scalable and efficient manner, such as the SGI Origin 2000 [74]. Since then, latency/bandwidth tradeoffs have brought a broad variety of coherence proposals such as those that follow a broadcast-based (snooping) approach, or the point-to-point based (directory) cache coherency protocols. However, in the context of many-core CMPs, different technological parameters and constraints must be considered. For example, cache-to-cache miss latencies are relatively shorter and the on-chip bandwidth is much larger than for the "off-chip" systems of the 90s. On the other hand, design decisions are severely constrained by power dissipation.

As we will expose in Chapter 5, the design of an efficient coherence protocol for shared-memory many-core CMPs should take into account several aspects related to efficiency such as on-chip area overhead, energy consumption, and performance. Nevertheless, another important metric to be considered is its resulting complexity. In this thesis, we propose an efficient and simple coherence protocol to meet with those requirements at once.

1.6 The *G-Lines* Technology

The continuous improvements in CMOS transistors following the Moore's Law will come to a stop in the near future because of several challenges: unsustainable power dissipation, physical limits of transistors reaching atomic scale dimensions, process and device variability, no real performance increases with scaling, and expensive R&D and manufacturing costs. To overcome that, chipmakers have recently begun to reconsider the *More-than-Moore* trend [150], where added value to devices is provided by incorporating functionalities that do not necessarily scale according to the Moore's Law. That is the reason why the future roadmap for semiconductor technology integrates both digital and non-digital technologies in the same chip [56]. Examples for the latter technology are: analog circuits, RF, optical technology, etc.

In this thesis, we make use of a state-of-the-art analog technology, namely *Global Lines* technology, or *G-Lines* from now on. *G-Lines* have already been successfully integrated in a silicon substrate in order to enable speed-of-light point-to-point communications. Chang et al. [126] and Jose et al. [9] showed early point-to-point circuits allowing transmission-line, wave-like velocity for 10 mm of interconnect. Nonetheless, this initial implementation suffers from significant overheads in terms of power dissipation and die area. A great effort has been devoted to overcome such limitations. For instance, Ito et al. [48] extended *G-Lines* to support broadcast, multi-drop and bidirectional transmissions. This contribution enables both low-latency and multi-drop ability on a transmission line with low-power dissipation. However, their results still exhibit several integration density issues. Additionally, Ho et al. [122] and Mensink et al. [39] have shown that a capacitive feedforward method of global interconnect reduces both power dissipation and die area overheads. In particular, they achieve nearly single-cycle delay for long wires with voltage-mode signaling. As a result, every *G-Line* is basically a shared wire that broadcasts 1-bit messages (signals from now on) across one dimension of the chip in a single clock cycle. A practical use of *G-Lines* is presented by Krishna et al. [142] in the context of networks-on-chip (NoC). Krishna et al. leveraged *G-Lines* using multi-drop connectivity and the *S-CSMA* collision detection technique to enhance a flow control mechanism (EVC) in terms of latency and power dissipation. In particular, these *G-Lines* are used to broadcast the control signals of EVC in order to communicate the availability of free buffers and virtual channels much more accurately. Furthermore, the authors employ the *S-CSMA* technique to calculate how many virtual channels or free buffers are demanded at any time in order to grant requests accordingly.

As we will see, in this thesis we also leverage this technology to deploy dedicated *G-Line*-based networks on chip, in order to implement synchronizations and coherence protocols to overcome the previously described performance limitations in future many-core CMPs.

1.7 Thesis Motivation and Contributions

Many-core CMPs demand new solutions to keep up with the exponential growth in performance as historically has been made and predicted. The previous sections have exposed some of the major actual performance limitations which are present in this kind of systems that we have categorized in synchronization and coherence problems (Sections 1.4 and 1.5). We have also considered to make use of special technology in Section 1.6 that, due to its extremely fast nature and minimal area overhead and power dissipation, helped us provide very efficient hardware-based implementations to overcome such performance limitations.

The main contributions of this thesis are summarized as follows:

- An *efficient hardware-based barrier implementation*, namely *GBarrier*, based on two main components. First, a very lightweight dedicated on-chip network that could be deployed in a hierarchical layout for scalability. And second, a very simple synchronization protocol. By levering both full-custom *G-Lines* technology and a nowadays standard toolflow to implement our proposal, we have found significant performance improvements for a simulated many-core CMP in terms of execution time, network traffic and energy consumption in comparison to the most efficient software-based barrier to date. Finally, we conclude that our proposal is not so dependent on full-custom technology to achieve significant improvements, by requiring negligible on-chip area and minimal power dissipation for both technologies.

- An *efficient hardware-based infrastructure for highly-contended locks*, namely *GLock*, which deploys a dedicated on-chip network and relies on a simple token-based messaging-protocol in order to provide an extremely efficient and completely fair lock implementation. To implement *GLock*, we have also considered two technologies: full-custom *G-Lines* technology and standard technology. Independently of the type of technology used, our proposal achieves significant improvements in execution time, network traffic and energy consumption when comparing this metrics with respect to the

most efficient software-based implementation for highly-contended locks. Moreover, *GLock* requires minimal on-chip area overhead and negligible power dissipation for both technologies.

- A *simple and very efficient cache coherence protocol*, namely *ECONO*, is designed for future many-core CMPs. In ensuring coherence, our proposal relies on express coherence notifications which are broadcast atomically over a dedicated lightweight on-chip network leveraging *G-Lines* technology for superior efficiency. While our protocol meets simplicity in its design, a performance comparison in a simulated many-core CMP platform against two contemporary coherence protocols, brings about a high efficiency in terms of execution time, network traffic, and is the most energy efficient design.

All the contributions of this thesis have been published in national conferences [66], [70], relevant international peer reviewed conferences [63], [65], [64], [67], [71] and relevant peer reviewed journals [69], [68]. Moreover, *ECONO* is currently being considered for publication in an international peer reviewed conference.

1.8 Thesis Organization

The rest of this thesis is organized as follows:

- Chapter 2 discusses the many-core CMP architecture considered, delves into the simulation tools employed, and gives a description of the workloads and metrics used throughout the thesis.
- Chapter 3 presents and evaluates our *GBarrier* proposal for barrier synchronization in many-core CMPs.
- Chapter 4 presents and evaluates our *GLock* proposal for highly-contended locks in many-core CMPs.
- Chapter 5 presents and evaluates our *ECONO* proposal for a simple and efficient coherence protocol in many-core CMPs.
- Chapter 6 summarizes the main conclusions of the thesis and points out future lines of work.

Chapter 2

Evaluation Methodology

This chapter presents the experimental methodology that we have followed throughout this thesis to implement and evaluate our proposals. For that, we start in Section 2.1 by describing the target system that we based on to incorporate our hardware optimizations. The simulation tools used are presented in Section 2.2 The descriptions of the benchmarks running on top of the simulation tools appear in Section 2.3. Finally, Section 2.4 discusses the metrics and methods to quantify the performance benefits that our proposals achieve and, at the end of this section, we summarize in a table the main characteristics of this evaluation methodology.

2.1 Target System

We choose a tiled CMP design as reference because its modular nature has made it popular in several commercial many-core designs [53, 54, 121]. The basic architecture is designed as arrays of identical or close-to-identical building blocks knowns as tiles, overlaid over a point-to-point interconnect typically forming a mesh-based network-on-chip. From now on, this kind of system will be referred to as tile-based many-core CMP, or simply many-core CMP.

An example of this kind of system is shown in Figure 2.1 for a 3×3-core CMP with a 2D-mesh layout. We can observe that each tile has a processing core, one level of private cache, a slice of the level-two cache along with its directory entries, and some routing logic. Considering that part of the appeal of CMPs is their ability to exploit TLP and provide higher throughput than a wider-issue uniprocessor while consuming less energy per operation, we have modeled the

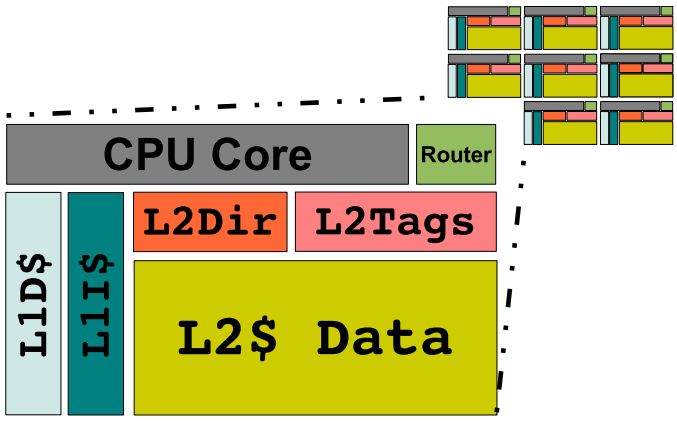


Figure 2.1: 3×3-core CMP architecture with a 2D-mesh topology.

processing cores of our CMP architecture after lightweight, in-order processors. Split instruction and data caches are available at the private level, while the second level is unified, physically distributed but logically shared amongst all processing cores. Private caches are kept coherent across the unordered network through an on-chip distributed directory protocol. The L1 caches maintain inclusion with the L2 cache, trading off some on-chip capacity for lower design complexity in the coherence controllers. Moreover, we use the less significant bits of the block address to determine the home tile for every memory block.

While our proposals have been specifically designed for a many-core CMP, different settings for this basic system have been evaluated in function of the type of simulation tool employed as well as the optimization proposed. This is the reason why we defer the specific details of each simulated system to the chapter where the particular proposal is described, optimized and evaluated.

2.2 Simulation Tools

As our proposals have their own idiosyncrasies and involve different optimizations in the context of many-core CMPs, we have chosen two different simulation tools: Sim-PowerCMP, to implement *GBarrier* and *GLock*; and Simics-GEMS, to implement *ECONO*. Moreover, we utilize a toolflow to determine efficiency using a current industrial standard cell design methodology.

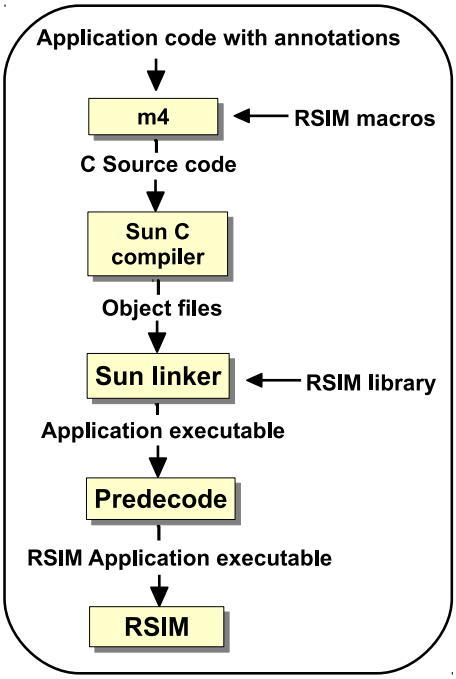


Figure 2.2: How parallel applications are transformed for simulation in Sim-PowerCMP.

2.2.1 Sim-PowerCMP

Sim-PowerCMP [3] is a detailed architecture-level power-performance simulation tool that models tiled-CMP architectures with a shared L2 cache on-chip and a MESI directory-based cache coherence protocol. Particularly, this simulator is based on a Linux x86 port of RSIM [24] and models a CMP architecture consisting of arrays of replicated tiles connected over an on-chip network. Each tile contains a processing core with primary caches (both instruction and data caches), a slice of the L2 cache, and a connection to the on-chip network as shown in Figure 2.1. Sim-PowerCMP estimates power dissipation by implementing already proposed and validated power models for both dynamic power (from Wattch [27], CACTI [52]) and leakage power (from HotLeakage [155]) of each processing core (including the L1 caches), the shared multi-bank L2 cache, as well as the interconnection network (from Orion [49]).

To run parallel applications on top of the Sim-PowerCMP performance simulator, we have to follow a particular process that transforms a parallel application code into a runnable code for this simulator. It is shown in the block diagram

illustrated in Figure 2.2. As shown in the figure, the first step is to write the applications in `C` with annotations that add parallel shared memory semantics to `C`. In all cases, the programs use the annotations proposed by Boyle et al. [58] (usually known as PARMACS macros). In consequence, the *m4* preprocessor transforms the parallel programs into plain `C` sources substituting the parallel shared memory annotations into `C` statements or suitable library calls, using the set of PARMACS macros included in the RSIM distribution. We must consider that, like RSIM, Sim-PowerCMP simulates applications compiled and linked for SPARC V9/Solaris using ordinary SPARC compilers and linkers. In this way, the Sun SPARC `C` compiler for Solaris is used to generate the relocatable object files that the Sun linker links together (along with the RSIM library, `C` library and math library), to create an executable SPARC application. For faster processing and portability, the Sim-PowerCMP simulator actually interprets applications in an expanded, loosely encoded instruction set format. For that, a predecoder is finally used to convert the executable SPARC application into this internal format, which is then fed into the simulator.

This performance simulator has been utilized in this thesis to implement and evaluate the two synchronization mechanisms introduced in the previous chapter: *GBarrier* and *GLock*. For the implementation, we have extended both the RSIM macros and the RSIM library to make visible at application-level code the procedure calls required to use *GBarrier* and *GLock*. In addition, we have extended the kernel of the Sim-PowerCMP to integrate the functionality provided by them. For the evaluation part, a number of relevant multi-threaded benchmarks listed in Section 2.3 have been conveniently transformed as aforementioned. After running the benchmarks, the detailed statistic reports of this simulator allowed us to determine the exact magnitude of the performance benefits that the use of our proposals entail. Later on, in Section 2.4, we describe the metrics, among all those reported by Sim-PowerCMP, that we focused on in our evaluations. Finally, our proposals have been evaluated against some of the software-based synchronization primitives provided by RSIM. In this way, in case of *GBarrier*, we have made use of a sophisticated binary combining-tree barrier [29]. In regards to *GLock*, we have employed a `test-and-test&set` implementation [29] for locks with low contention, and we have extended the RSIM library with the most efficient software-based implementation for highly-contended locks considered to date (*MCS Locks* [76]).

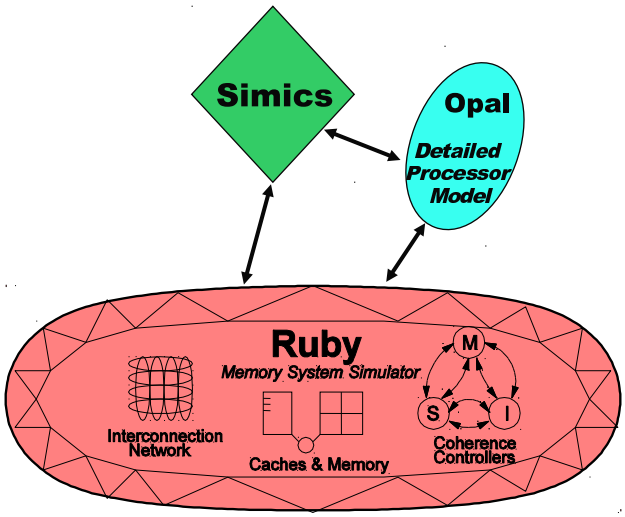


Figure 2.3: Architecture of the Simics-GEMS simulation framework.

2.2.2 Simics-GEMS

We have also used the Wisconsin *General Execution-driven Multiprocessor Simulator* (GEMS) simulation environment [98], which is based on Wind River Simics [118]. Simics is a full-system functional simulator of multiprocessor systems that supports the SPARC instruction set architecture (ISA) amongst others, on which we run a number of parallel benchmarks described in Section 2.3. In particular, we use Simics 3.0.31 to boot an unmodified Solaris 10 box with 16 processors and 4GB of memory, using the *sarek* target machine (Sun Fire 6800 server with UltraSPARC-III Cu processors). Simics supplies an in-order processor model in which all instructions take one cycle to execute. Simics then allows an external module to register with its timing interface, so that the latency of memory access instructions can be modeled with accuracy.

While Simics is responsible for the functional correctness of the simulation framework, GEMS provides several timing modules that plug into Simics to incorporate detailed models for the fundamental components of the system. In particular, we used GEMS 2.1 to implement our *ECONO* proposal. As shown in Figure 2.3, GEMS comprises two main modules, namely *Ruby* and *Opal*. The former models memory hierarchies and uses *Specification Language for Implementing Cache Coherence* (SLICC), a domain-specific language to describe the behaviour of coherence protocols in terms of their state machine. The latter captures the

temporal features of an out-of-order processor. A third module called *Tourmaline* allows Simics to be used as functional simulator for transactional applications, enabling near-Simics execution speeds while preserving transactional semantics. For the evaluations presented in this thesis, we model a chip-multiprocessor (CMP) composed by simple in-order processing cores [8], and hence only the Ruby module was used.

The Ruby module offers an event-driven framework to precisely simulate a memory hierarchy that allows us to measure the effects of behavioral and structural changes to the components that conform the memory subsystem, namely L1 and L2 caches, and directory and memory controllers. Ruby models the latency of each memory request received from the functional simulator by stalling Simics until the memory hierarchy brings the requested data with appropriate permissions to the requesting processor's first level cache. As a request travels across the memory subsystem, each component introduces a given delay, measured from the moment the message is picked from its input port for processing, until the component generates a response and injects it back into the network. All the components are connected using a detailed network model called *Garnet* [107], which features a state-of-the-art interconnect that precisely models the time required to deliver a message from one component to another.

Simics-GEMS was employed to implement our last proposal of this thesis: a cache coherence protocol called *ECONO*. We decided to make use of this simulation tool because SLICC considerably simplifies and speeds up the development of the cache and directory controllers for *ECONO* and the different protocols discussed in this thesis. Moreover, this simulation tool also provides a very detailed report of coherence-related statistics that helped us gain insight into where efficiency and performance degradation come from in every case.

2.2.3 Mainstream Industrial Toolflow

A *Mainstream Industrial Toolflow* was also used to precisely quantify the operation timing and on-chip area overhead that our proposals feature when considering a current industrial standard cell design methodology.

Figure 2.4 outlines a block diagram of the toolflow utilized in this thesis. There are six main boxes that represent the six common steps performed in a current mainstream industrial toolflow from design specification to silicon fabrication. As we can see on the right part of the figure, we also illustrate the tools that allowed us to accomplish the steps in the flow they are attached to (e.g. ModelSim is the tool for functional verification).

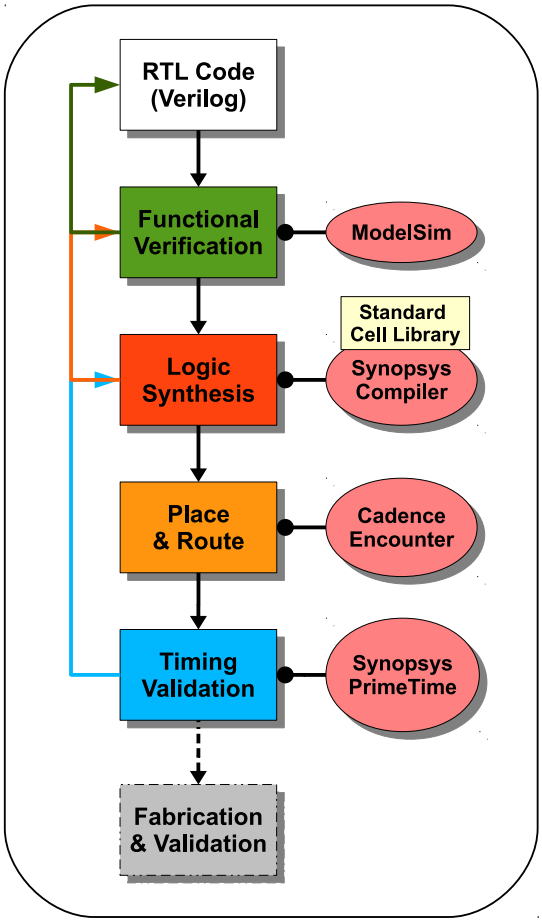


Figure 2.4: Mainstream industrial toolflow.

The first step involves the codification of the design we want to physically implement or synthesize. The language chosen for this step is the *Hardware Description Language* (or HDL) called Verilog [148]. In particular, we describe our digital system following a behavioral model rather than the lower-level *Register Transfer Language* (RTL) model shown in the figure. The RTL code is very close to assembly language based on describing how data is transformed by pure combinational logic as it is passed from register to register[1]. In contrast, the behavioral code looks very similar to `C`, declaring procedural blocks where both

[1]The term register refers to a sequential logic component, like a Flip-Flop, which stores an input or an output value produced during a given clock cycle by the combinational logic part of the designed system.

the combinational and the sequential codes are written. From the programmer standpoint, the main difference with respect to an imperative paradigm language such as `C` is that all blocks in the Verilog code are executed concurrently. This is due to the fact that every block models a component of the designed system (e.g. a counter) that runs independently because it is fed with one of the leaves of the clock tree signal (i.e. a different leaf for every block). In Verilog there are two kind of blocks that we must deal with. The *always* block, which, as the name suggests, is executed in a loop over and over again and is used to encapsulate the functionality of a particular component of the system. An *always* block has input and output ports to establish an interaction with the block or they can be used to communicate with other blocks. The other type is the *initial* block, which is executed only once at time zero and it commonly initializes the clock and reset signals, and gives the initial values to the input ports that feed the *always* blocks.

The second step comprehends a functional verification of the previous Verilog codification for the design in order to know if all of its blocks work correctly. For that a testbench is written which generates a clock, reset a set of input signals that stimulate all the blocks (i.e. it is the *initial* block) and gives a report of all the output signals. The ModelSim [104] is the tool that we chose for this step, which compiles the Verilog code at behavioral level, including the testbench, to generate the RTL code that is simulated to carry out the functional verification. This tool was chosen since it includes a powerful graphical user interface that easily allowed us to check all the input and output values on a cycle-by-cycle basis (even a more precise timing scale can be watched) in order to know if the design meets the timing requirements (i.e. it generates the right output values at the right moment). If the design is not correct at this level, we must go back to the first step in the toolflow (see the figure) and iterate until a correct design is obtained.

Once our design is correct at level of RTL code, we have to synthesize and to check after synthesization whether the tool in charge of the synthesization has found any error in the design (e.g. the combinational part has logic that never produces a value, or there are some unconnected ports, or the tool has produced Latches to implement some registers rather than using the more predictable Flip-Flops). To this end, we used the Synopsys Physical Compiler [139] synthesis tool that takes as inputs the RTL code and the target technology library by which we want to synthesize our design, and produces a preliminary synthesized design. In particular, we rely on a 45 nm standard cell technology library provided by STMicroelectronics [138]. As a result of this step, the RTL code is mapped to a set of basic logic components or cells (e.g. AND gates or D-Type Flip-Flops) and the

tool automatically inserts the necessary RC-based wires to interconnect these cells, in function of the input/output port connections declared at Verilog codification. In particular, we use the `dc_shell` program provided by the Synopsys Physical Compiler. This program can be configured with an specific clock period by which the resulting synthesizable design must operate. Moreover, after a successful synthetization process, it produces a cell-based design (netlist) along with a detailed report where we can see if the design can meet a particular constraint called slack. That is, the slack will be met as long as the time of the longest path in the netlist between two registers in the design is less or equal to the given clock period. Otherwise, we can either increase the clock period or re-design the Verilog code by, for example, including additional registers between the longest path to minimize its latency. It is important to note that at this stage of the toolflow, the tool is not aware of wire delays and the timing is estimated based on the cell delays.

The netlist from the synthesis tool has to be properly placed in a given floorplan (i.e. where cells belonging to a particular component of the design must be placed on chip). Moreover, distance-aware connections between those components have also to be established in order to obtain minimal propagation delays for the signals that travel over these RC-based wire connections. To this end, we used the Cadence SoC Encounter Tool [26]. This tool takes as input the dimensions of the floorplan along with a topographical placement of the components of our design to instruct where to place each of them in the floorplan. Then, the tool estimates the shortest connections for the placement constraints indicated to the tool. Moreover, the tool also routes the clock tree and the reset signals that input all components in the design. For our designs, we assume a single clock domain with a unique clock tree for the whole many-core CMP. Apart from the netlist enriched with all the RC-based wires (post-layout design), this tool also dumps out a detailed description of timing for all components in the system including the parasitic side effects that degrade propagation latencies for the connections between components. In addition, this tool also provide a detailed report of the on-chip area overhead required by the synthesized post-layout design.

The next step involves checking if the timing properties of the post-layout design meets or not the timing constraints of the initial design. For that, a timing validation is executed using a sign-off procedure performed by the Synopsys PrimeTime tool [140] using its `pt_shell` program. If the timing constraints are met, the design is ready to be directly manufactured by the industry, and it must be also tested after fabrication to detect any possible bug in the design. Otherwise,

a re-design of the system or another adjustment in the timing constraints followed must be done by going back to earlier steps in the toolflow.

To conclude this section, we want to justify the use of this Mainstream Industrial Toolflow in this thesis. The main reason was to determine whether our proposals are fully dependent on a full-custom non-standard technology to work efficiently in terms of minimal operation latency and negligible on-chip area overhead. As we will explain, while *GBarrier* and *GLock* could be successfully implemented using the previously described standard methodology, *ECONO* requires the use of a full-custom technology to be efficient enough for its integration in many-core CMPs.

2.3 Benchmarks

In this section, we describe the benchmarks that we have considered in this thesis to evaluate our proposals. It is important to note that each of our proposals has been evaluated by utilizing the particular subset of the benchmarks that better reflect the characterization to be done. For instance, benchmarks that show a significant fraction of their execution time devoted to lock synchronizations were employed to evaluate our hardware-based lock proposal (*GLock*). Here, we distinguish three types of benchmark: Microbenchmarks, Kernels and Real or Scientific applications. Moreover, a number of synthetic benchmarks were also used in an attempt to gain insight into the potential benefits that our proposals could entail, although to have a clear understanding of their use, they have not been included in this section so that their descriptions will be given later in the corresponding chapters.

2.3.1 Microbenchmarks

To understand the basic performance capability of a particular design, microbenchmarks are commonly used. In particular, we have implemented five different microbenchmarks following a methodology similar to the one used in [125, 130].

Single Counter (SCTR) consists of a counter (fits in a cache line), protected by a single lock, that is incremented by all threads in a loop.

Multiple Counter (MCTR) is made up of an array of counters (residing in different cache lines), protected by a single lock, where each thread increments a different counter of the array in a loop.

Doubly Linked list (DBLL) builds a doubly linked list, protected by a single lock, where threads dequeue elements from the head of the list and enqueue them into the tail of the list afterwards.

Producer Consumer (PRCO) consists of a shared FIFO (bounded) array, protected by a single lock, that is initially empty. Half the threads enqueue items into the FIFO that are consumed by the other half of threads. Producers have to wait for free slots in the FIFO whereas consumers have to wait for data to consume before iterating the critical section code.

Affinity Counter (ACTR) uses two locks that protect two counters accessed consecutively by all threads. For each iteration, all threads acquire the first lock to update the first counter, the barrier synchronizes them, and then the second lock is acquired to modify the second counter.

2.3.2 Kernels

Several kernels have also been considered in this thesis. This kind of workload extract some well-defined parts of complete applications to reason about achieved performance by focusing on a particular behavior of an specific group of operations in the code. For instance, to evaluate the performance benefits derived from our *GBarrier* implementation, a piece of application' code that is barrier intensive would constitute a Kernel.

FFT. The *FFT* kernel is a complex one-dimensional version of the radix-$\sqrt{n}$ six-step FFT algorithm, which is optimized to minimize interprocessor communication. The data set consists of the n complex data points to be transformed, and another n complex data points referred to as the roots of unity. Both sets of data are organized as $\sqrt{n} \times \sqrt{n}$ matrices partitioned so that every processor is assigned a contiguous set of rows which are allocated in its local memory. All synchronization operations in the FFT's code are implemented by means of a few barriers that do not represent a relevant fraction of the total execution time of this kernel.

Livermore Loops. Livermore loops [41] have long been used as a tough test for compilers and architectures. They present a wide array of challenging kernels where fine-grain parallelism is present but it is hard to extract and exploit efficiently. Following the recommendations given in [82], we have focused on Kernels 2, 3 and 6 because they present a significant amount of barrier operations in their codes. In short, Kernel 2 is an excerpt from an incomplete Cholesky conjugate gradient code. Kernel 3 is a simple inner product. And finally, Kernel 6 is a general linear recurrence equation.

Radix. The *Radix* kernel sorts a series of integers, called keys, using the popular radix sorting method. The algorithm is iterative, performing one iteration for each radix r digit of the keys. In each iteration, a processor passes over its assigned keys and generates a local histogram. The local histograms are then accumulated into a global histogram. Finally, each processor uses the global histogram to permute its keys into a new array for the next iteration. This permutation step requires all-to-all communications by using barriers which do not represent a significant fraction of the kernel execution time. The permutation is inherently a sender-determined one, so keys are communicated through writes rather than reads.

2.3.3 Scientific and Real Applications

Finally, we have chosen several scientific and real applications belonging to different benchmark suites and others. From SPLASH-2 [128]: Barnes, Ocean, Radix and Raytrace. Moreover, belonging to PARSEC benchmark suite [18], we chose Swaptions. Other benchmarks in this group are: EM3D, Tomcatv, Unstructured and QSort.

Barnes. The *Barnes* application simulates the interaction of a system of bodies (galaxies or particles, for example) in three dimensions over a number of time steps, using the Barnes-Hut hierarchical N-body method. Each body is modeled as a point mass and exerts forces on all other bodies in the system. To speed up the interbody force calculations, groups of bodies that are sufficiently far away are abstracted as point masses. In order to facilitate this clustering, physical space is divided recursively, forming an octree. The tree representation of space has to be traversed once for each body and rebuilt after each time step to account for the movement of bodies. The main data structure in Barnes is the tree itself, which is implemented as an array of bodies and an array of space cells that are linked together. Bodies are assigned to processors at the beginning of each time step in a partitioning phase. Each processor calculates the forces exerted on its own subset of bodies. The bodies are then moved under the influence of those forces. Finally, the tree is regenerated for the next time step. There are several barriers for separating different phases of the computation and successive time steps. Some phases require exclusive access to tree cells and a set of distributed locks is used for this purpose. The communication patterns are dependent on the particle distribution and are quite irregular. No attempt is made at intelligent distribution of body data in main memory, since this is difficult at page granularity and not very important to performance.

EM3D. This benchmark is a shared memory implementation of the Split-C benchmark [28]. The *EM3D* application models the propagation of electromagnetic waves through objects in three dimensions. The problem is framed as a computation on a bipartite graph with directed edges from E nodes, representing electric fields, to H nodes, representing magnetic fields, and vice versa. At each step in the computation, new E values are first computed from the weighted sum of neighboring H nodes, and then new H values are computed from the weighted sum of neighboring E nodes. Edges and their weights are determined statically. The initialization phase of EM3D builds the graph and does some precomputation to improve the performance of the main loop. To build the graph, each processor allocates a set of E nodes and a set of H nodes. Edges are randomly generated using a user-specified percentage that determines how many edges point to remote graph nodes. The sharing patterns found in this application are static and repetitive. Synchronization in this application is accomplished by using barriers.

Ocean. The *Ocean* application studies large-scale ocean movements based on eddy and boundary currents. The algorithm simulates a cuboidal basin using discretized circulation model that takes into account wind stress from atmospheric effects and the friction with ocean floor and walls. The algorithm performs the simulation for many time steps until the eddies and mean ocean flow attain a mutual balance. The work performed every time step essentially involves setting up and solving a set of spatial partial differential equations. For this purpose, the algorithm discretizes the continuous functions by second-order finite-differencing, sets up the resulting difference equations on two-dimensional fixed-size grids representing horizontal cross-sections of the ocean basin, and solves these equations using a red-back Gauss-Seidel multigrid equation solver. Each task performs the computational steps on the section of the grids that it owns, regularly communicating with other processes. Synchronization is performed by using both locks and barriers.

QSort. The *Qsort* [17] application is a well-known sorting algorithm that is based on the principle of resolving a problem into two simpler subproblems. Each of these subproblems may be resolved to produce yet simpler problems. The process is repeated until all the resulting problems are found to be trivial. These trivial problems may then be solved by known methods, thus obtaining a solution of the original more complex problem. Sorting a given array of integers require many frequent lock operations to protect the integrity of the data structure for every modification done in the items to be sorted.

Raytrace. This application renders a three-dimensional scene using ray tracing. A hierarchical uniform grid is used to represent the scene, and early ray termination is implemented. A ray is traced through each pixel in the image plane and it produces other rays as it strikes the objects of the scene, resulting in a tree of rays per pixel. The image is partitioned among processors in contiguous blocks of pixel groups, and distributed task queues are used with task stealing. The data accesses are highly unpredictable in this application. Synchronization in Raytrace is done by using locks. This benchmark is characterized for having very short critical sections and very high contention. Barriers are not used for the Raytrace application.

Swaptions. The Swaptions application is an Intel workload which uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions. The HJM framework describes how interest rates evolve for risk management and asset liability management for a class of models. Because HJM models are non-Markovian the analytic approach of solving the PDE to price a derivative cannot be used. Swaptions employs Monte Carlo (MC) simulation to compute the prices. To do so, the program stores the portfolio in a swaptions array. Each entry corresponds to one derivative. Swaptions partitions the array into a number of blocks equal to the number of threads and assigns one block to every thread. Each thread iterates through all swaptions in the work unit it was assigned and calls the function HJM Swaption Blocking for every entry in order to compute the price. In this application there is a small fraction of execution time due to synchronization operations comprised of a few locks.

Tomcatv. This is a parallel version of the SPECCFP95's tomcatv application [136]. The original source was part of Prof. W. Gentzsch's benchmark suite. *Tomcatv* is a highly vectorizable program for the generation of two-dimensional boundary-fitted coordinate systems around general geometric domains such as airfoils, cars, etc. It is based on the method introduced in 1974 which uses two Poisson equations to produce a mesh which adapts to the physical region of interest. The transformed non-linear equations are replaced by a finite difference approximation, and the resulting system is solved using successive live overrelaxation.

Unstructured. Unstructured is a computational fluid dynamics application that uses an unstructured mesh to model a physical structure, such as an airplane wing or body. The mesh is represented by nodes, edges that connect two nodes, and faces that connect three or four nodes. The mesh is static, so its connectivity does not change. The mesh is partitioned spatially among different processors using a recursive coordinate bisection partitioner. The computation contains a

series of loops that iterate over nodes, edges and faces. Most communication occurs along the edges and faces of the mesh. Synchronization in this application is mainly accomplished by using barriers.

2.4 Metrics and Methods

The evaluation of the proposals presented in this thesis has been carried out using different metrics depending on the particular optimization we are providing. While each of our proposals have required a particular set of metrics, in general we have considered the following: performance, network traffic, on-chip area overhead and power dissipation. The other metrics specific to each proposal will be discussed where appropriate.

For evaluating the performance, we measure the total number of cycles employed for each application during its parallel phase, i.e., the execution time of the parallel phase. Although the IPC (instructions per cycle) constitutes a common metric for evaluating performance improvements, it is not appropriate for multithreaded applications running on multiprocessor systems [10]. This is due to the spinning performed during the synchronization phase of the different threads. For example, a thread can be repeatedly checking the value of a lock until it becomes available, which increases the number of completed instructions (and maybe the IPC) but actually the program is not making progress. On the one hand, for our two synchronization proposals (*GBarrier* and *GLock*), in order to better understand the reasons why we improve performance, we depict a breakdown of the applications' execution time in order to account for the fraction of the execution time devoted to synchronization, memory operations and effective computation (i.e. arithmetic operations). On the other hand, for our coherence protocol (*ECONO*), apart from estimating the execution time improvements, we also analyze the coherence activity performed and assess the reduction in L1 cache misses achieved considering each miss individually. By doing so, we precisely understand where the improvements come from. Moreover, another important metric here to take into account is the number of protocol hops. Note that, performance improvements can also be a consequence of the number of coherence messages that are needed in the critical path of a cache miss for solving it.

We also measure the traffic injected in the interconnection network by the three proposals and, when required to reason about improvements, we expose a decomposition of the traffic in function of the magnitude of every type of message

transmitted: data and control messages (requests and coherence activity). In case of the control messages 8 bytes are required while 72 bytes are employed for data messages.

Regarding the on-chip area overhead, we show both detailed and qualitative analysis of this metric. In the former case, we employ the statistics reported by the mainstream industrial toolflow presented in Section 2.2.3 when synthesizing our proposals. The latter case is a valid option if there is a very clear evidence that the magnitude of the area overhead is negligible, so that a precise quantification is unnecessary. That is the case when using the full-custom technology of Section 1.6 to implement our proposals. As we will see, if marginal area numbers are estimated for a much costlier design (the on-chip network utilized in the work where this technology is explained for the first time [143]), obviously our proposals will require negligible on-chip area overhead.

We have also provided results in terms of the power dissipated by our proposals. As for the area metric, we show both detailed and qualitative outcomes, although in this case the detailed analysis is done for the full-custom technology. First, we estimate the power dissipation relying on the power parameters for a 65-nm CMOS process simulated in [143]. And second, in case of our synchronization proposals, we integrate the already commented estimated power dissipation in the simulation tool employed for them (Sim-PowerCMP), to obtain the total power dissipation statistics for the full CMP. By means of the latter results, we have also estimated the energy efficiency utilizing the energy-delay^2 product (ED^{2}P) metric. The negligible on-chip area overheads that our proposals entail when using the mainstream industrial toolflow led us to conclude the same for the power dissipation.

Finally, we describe some other important aspects related to the methodology used. All the experimental results reported in this thesis correspond to the parallel phase of each program. In case of Simics-GEMS, we have created checkpoints for every benchmark in which each application has been previously executed to ensure that memory is warmed up and, hence, avoiding the effects of page faults. Then, we run each application again up to the parallel phase, where each thread is bound to a particular core. The application is then run with full detail during the initialization of each thread before starting the actual measurements. In this way, we warm up caches to avoid cold misses. Moreover, the different extensions and protocols implemented for our *ECONO* proposal have been exhaustively checked using a tester program provided by GEMS. The tester program stresses corner cases of cache coherence protocols to raise any incoherence by issuing requests that simulate very contended accesses to a few memory blocks.

Table 2.1: Summary of the evaluation methodology.

Proposal	**Simulation Tools**	**Benchmarks**	**Metrics**
GBarrier	Sim-PowerCMP, Industrial Toolflow	Ocean, EM3D, Unstructured, Kernel 2, Kernel 3, Kernel 6	Time, Traffic, Area, Energy, Power
GLock	Sim-PowerCMP, Industrial Toolflow	Ocean, Raytrace, Qsort, SCTR, MCTR, DBLL, PRCO, ACTN	Time, Traffic, Area, Energy, Power
ECONO	Simics-GEMS	Barnes, FFT, Ocean, Radix, Raytrace, EM3D, Tomcatv, Unstructured, Swaptions	Time, Traffic, Area, Power, Cache Misses

To conclude this chapter, from all the discussions given above, Table 2.1 summarizes the evaluation methodology employed in this thesis.

Chapter 3

GBarrier: An Efficient Infrastructure for Barrier Synchronization

3.1 Introduction and Motivation

A barrier is a synchronization primitive that enables multiple processes or threads to wait in a particular point of execution, until all of them have reached it before any of them can continue. A typical example of its usage is utilizing barriers to separate the different phases commonly found in parallel applications [119]. By doing so, the programmer ensures that the second phase does not start until all processes or threads from the application have completed the first one.

In the context of systems that implement a shared-memory programming model [29], with the advent of multi-core architectures, new challenges are arising to provide an efficient barrier implementation. This is mainly due to the fact that differently to classical multiprocessor applications which target coarse-grained parallelism, multi-core applications tend to exploit fine-grained parallelism, and therefore, they may be highly sensitive to barrier performance [83].

Typical implementations of current software-based barriers (SW-barriers, from now on) rely on busy-waiting on shared variables which are atomically updated [76]. Nevertheless, the use of shared variables implies that the cache coherence protocol must come in on maintaining their consistency across all levels of the memory hierarchy. In turn, coherence activity translates into traffic injection in the interconnection network. As a result, an ever-growing amount of resources and energy may need to be devoted to support SW-barriers as the

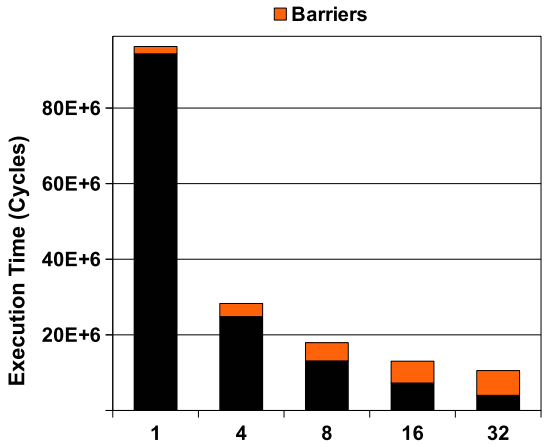


Figure 3.1: Fraction of time due to barriers in EM3D.

number of cores in many-core CMPs increases. On the other hand, busy-waiting on locally-cached shared variables has also significant implications on the energy consumed by the L1 caches.

As an example, Figure 3.1 illustrates the potential performance losses suffered in the EM3D parallel application when using SW-barriers in future many-core CMPs (for details about the evaluation see Section 3.4). In particular, we present the results obtained for a sophisticated binary combining-tree barrier (which is considered one of the most efficient SW-barriers) as the number of cores is increased from 1 to 32. Each bar shows the fraction of the execution time due to barrier synchronization in orange color. As can be derived from the figure, as the number of cores increases so does the fraction of the execution time due to barrier synchronization (up to 63% for 32 cores), thereby considerably limiting scalability.

In this chapter, we describe and evaluate an efficient barrier synchronization mechanism specifically designed for many-core CMPs. Differently from SW-Barriers, our proposal, namely *GBarrier*, has been implemented entirely in hardware. To implement *GBarrier*, we have explored two different technologies. On the one hand, we make use of the state-of-the-art full-custom *G-Lines* technology and the *S-CSMA* technique explained in Section 1.6. In short, every *G-Line* enables almost speed-of-light 1-bit communications across one dimension of the entire chip, and the *S-CSMA* technique is employed to detect the number of simultaneous transmissions over a *G-Line*. On the other hand, we utilize the mainstream industrial toolflow with standard cells in an advanced 45 nm process

(*Standard* technology from now on) presented in Section 2.2.3, in order to obtain a cost-effective implementation for our proposal at the expense of some negligible performance losses.

To show the benefits derived from *GBarrier*, we integrate both *GBarrier* implementations into the Sim-PowerCMP performance simulator (further details in Section 2.2.1) for a 32-core CMP layout. Moreover, we compare their performance results against the most efficient SW-barrier considered to date (a binary combining-tree barrier) by using several kernels and scientific applications. In particular, performance results for the *G-Lines* technology show an average reduction of 54% and 21% in execution time, for the kernels and scientific applications respectively. The latter performance results suffer from a penalization of 6.3% and 4.3% when using the other slower *Standard* technology, respectively. Nonetheless, as we will see, the relative improvement provided by *G-Lines* can be considered negligible in comparison to the much higher execution times reported by the most efficient SW-barrier implementation. In addition, given the fact that *GBarrier* does not deal with the main data network, it exhibits an average reduction of 53% and 18% in network traffic, for the kernels and scientific applications respectively. This traffic reduction also leads to an average reduction of 76% and 31%, for the kernels and scientific applications respectively, in the energy-delay^2 product (ED^{2}P) metric for the full CMP. Finally, we have also evaluated the area overhead and power dissipation that each technology-aware *GBarrier* implementation would entail, concluding that both of them are negligible regardless of the technology employed.

The rest of the chapter is organized as follows. We detail the architecture, operation, properties and costs of *GBarrier*, and discuss how our proposal can be generalized to operate on different scenarios in Section 3.2. Next, in Section 3.3 we discuss some important performance implications when using our proposal. Then, Section 3.4 shows the simulation environment used in this chapter and analyzes the benefits brought by *GBarrier* in terms of reductions in execution time, network traffic and power dissipation. Next, Section 3.5 discusses the related work on hardware barrier implementations. Finally, Section 3.6 presents our main conclusions.

3.2 The *GBarrier* Synchronization Mechanism

In this section we present our proposal to build an efficient hardware infrastructure for barrier synchronization in the context of many-core CMPs. To do so,

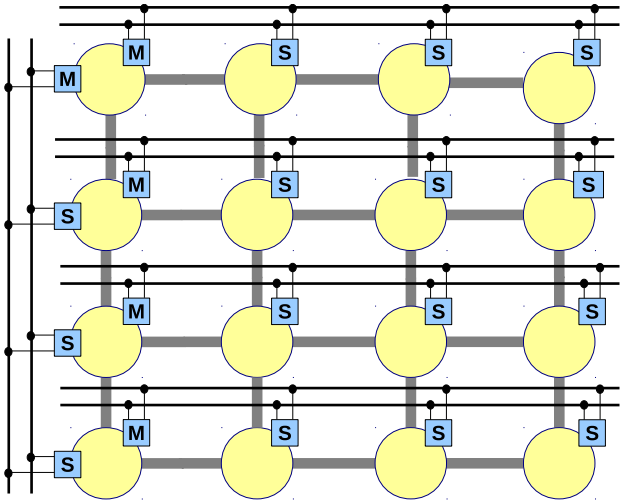


Figure 3.2: *GBarrier* architecture for a 16-core CMP with a 2D-mesh network.

we start by describing the architecture of the dedicated on-chip network that our proposal entails. For simplicity, the explanation will be given assuming the *G-Lines* technology with the *S-CSMA* technique. Later on, in Section 3.2.5 we discuss how *GBarrier* would be implemented using the *Standard* technology. As a case study, we choose a CMP with a 2D-mesh data interconnection network with R rows of C cores each (for a total of $N = R \times C$ cores), although our proposal is not restricted to this topology. Next, we show how the *GBarrier* mechanism would operate. After that, we describe the interface for programmers and provide details about the implementation of the set of controllers required by our proposal. Finally, we analyze the implementation costs and describe how our mechanism can be generalized to operate in several scenarios.

3.2.1 Dedicated On-Chip Network Architecture

The *GBarrier* mechanism relies on a dedicated on-chip network as it can be observed in the example in Figure 3.2. For simplicity, we concentrate on a version of the proposed network providing support for one barrier[1]. As shown in Figure 3.2, the *GBarrier* infrastructure is made up of two kind of components. *G-Lines* (horizontal and vertical finer black lines), that are used to transmit the signals required by the synchronization protocol; and controllers (M and S), that actually implement the synchronization protocol.

[1]In Section 3.2.6 we discuss the extensions to support several *GBarriers*.

As discussed in Section 1.6, every *G-Line* is a wire that enables the transmission of one bit of information across one dimension of the chip in a single clock cycle. Our *G-Line*-based network employs two *G-Lines* per barrier for every row and two more for the first column. In this way, for any 2D-mesh layout with R rows and C columns, the total number of *G-Lines* per barrier that would be needed is equal to $2 \times (R + 1)$ (e.g. 10 *G-Lines* for the 16-core CMP assumed in the example).

In addition to the *G-Lines*, our proposal also incorporates a set of controllers in charge of the synchronization protocol required for a barrier synchronization. In particular, we distinguish two types of controllers: master and slave controllers (see M and S in Figure 3.2, respectively). Each controller is attached to two *G-Lines*: one of them is used to transmit signals, whilst the other is employed to receive signals. More specifically, the *G-Line* used by the master controller to receive signals is the one used by the slave controllers to send signals, and vice versa. Moreover, the master controller is responsible for carrying out the count of signals transmitted from all slave controllers attached to the *G-Line*. To do so, the master controller contains a device that implements the *S-CSMA* technique. Recall that, this technique implements voltage amplitude sensing to determine the number of simultaneous transmitters over a particular *G-Line* at any given instant in time.

Finally, for design constraints [142] every *G-Line* can support up to six transmitters and one receiver as much, resulting in a CMP configuration with up to 7×7 cores. However, as we will explain in Section 3.2.6, *GBarrier* is not restricted to this number of cores and can efficiently operate with even larger core counts by means of a hierarchical architecture.

3.2.2 Synchronization Protocol

The synchronization protocol implemented on top of the *G-Line*-based network previously described relies on the exchange of 1-bit messages (signals) between the master and slave controllers, and the use of the *S-CSMA* technique in the master controllers to count the number of signals transmitted across every *G-Line*. In our proposal, every barrier synchronization is carried out by using a two-phase protocol: the *account phase* and the *release phase*. The first phase starts when the first thread arrives at the barrier and finishes when the last one reaches the barrier. Then, the second phase, in which all threads participating in the barrier are commanded to resume execution, is initiated. The exact interplay among threads, *G-Lines* and controllers is detailed below with an example.

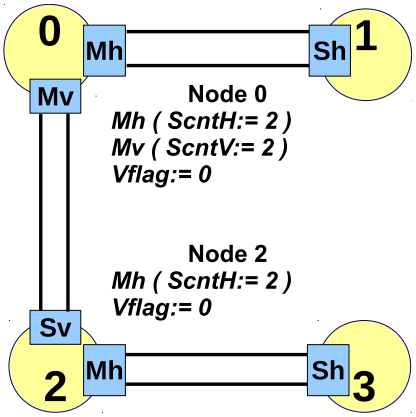


Figure 3.3: *GBarrier* for a 4-core CMP with a 2D-mesh network showing initial state of registers and flags.

Without loss of generality, we assume that all cores execute the same barrier at the same time and we explain how the barrier synchronization would take place on a 2×2 mesh layout (see Figure 3.3). We distinguish between horizontal and vertical controllers depending on the couple of *G-Lines* they are attached to. In this setting, there are four horizontal and two vertical *G-Lines*. Thus, there are two horizontal master controllers (see Mh in cores 0 and 2), two horizontal slave controllers (see Sh in cores 1 and 3), one vertical master controller (see Mv in core 0) and one vertical slave controller (Sv in core 2).

As shown in Figure 3.3, each master controller employs a couple of hardware elements during a barrier synchronization. The first is the *ScntH* and *ScntV* counters required by the horizontal and vertical master controllers respectively. These counters keep track of the number of signals (obtained through the *S-CSMA* technique) received from the horizontal slaves (cores 1 or 3) or vertical slaves (just one in this case), and whether the processor core the master controller is attached to has arrived at the barrier. The second element is the *Vflag* flag, which is used to establish a local synchronization between horizontal master controllers and the corresponding vertical controllers (master and slaves) located in the same core (Mh-Mv in core 0 and Mh-Sv in core 2). In particular, each *ScntH* counter is initialized with the number of slaves controllers in each row plus one to also account for the local core. *ScntH* is decremented every time a signal from a slave controller in its row is received through the corresponding *G-Line* (Sh in cores 1 and 3, in the example) and also when the local core arrives at the barrier. Once each *ScntH* counter reaches zero, the corresponding *Vflag* flag is set. Similarly,

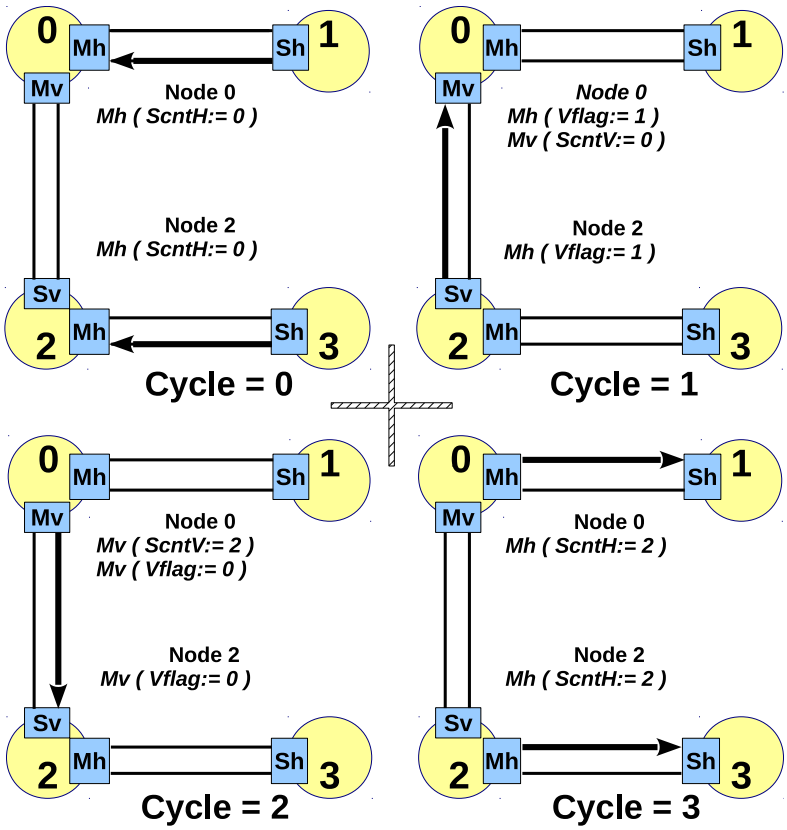


Figure 3.4: Barrier synchronization under *GBarrier*.

the initial value of the *ScntV* counter is the number of vertical slaves plus one, and is decremented on receiving a signal from a slave controller in its column (Sv in core 2, in the example) and when its local *Vflag* flag is set. It is worth noting that, as explained in Section 3.2.6.1, an initial setup is required in order to initialize both *ScntH* and *ScntV* counters to their maximum values. In the example of Figure 3.3, since all cores participate in the barrier, these counters will be initialized to two for both horizontal and vertical master controllers. From now on, these values will be referred to as *MAXH* and *MAXV* for the *ScntH* and *ScntV* counters, respectively.

Taking the initial setup shown in Figure 3.3 as the starting point, Figure 3.4 illustrates an example of how the barrier synchronization process would be performed. It is worth noting that we are assuming theoretical synchronization latencies that may not be reflected in the exact number of clock cycles required for the two physical *GBarrier* implementations (see Section 3.3.1).

At cycle 0, the *account phase* starts because all threads notify their arrival at the barrier. To do so, the horizontal slaves (Sh) signal, through their corresponding transmission lines, the arrival of cores 1 and 3 at the barrier. In turn, the horizontal masters decrement their *ScntH* counters with the number of received signals (*ScntH:=1*). Besides, each *ScntH* counter is also decremented to reflect the fact that cores 0 and 2 have also arrived at the barrier (see *ScntH:=0* in the figure). At cycle 1, once each horizontal master has detected that its local counter *ScntH* has reached zero, it sets its *Vflag* flag (*Vflag:=1*) in order to make the corresponding vertical slave (Sv) or master (Mv) controller to proceed with the vertical stage of the *account phase*. Then, the vertical slave (Sv) signals, through its corresponding transmission line, the arrival of cores 2 and 3 at the barrier and the vertical master (Mv) decrements its *ScntV* counter (*ScntV*:=1). Moreover, the *ScntV* counter is also decremented because the cores 0 and 1 have also arrived at the barrier and the *Vflag* flag was set (*ScntV*:=0). After the *ScntV* counter reaches zero, the *release phase* is initiated. To do so, at cycle 2, the vertical master unsets the *Vflag* flag (*Vflag:=0*), resets the local *ScntV* counter to its initial value (*ScntV:=2*) and signals the vertical slave, through the corresponding vertical *G-Line*. Upon reception of the signal, the vertical slave also resets the *Vflag* in order to make the horizontal masters to proceed with the horizontal stage of the *release phase*. At cycle 3, the horizontal masters initialize the local *ScntH* counters (*ScntH:=2*), and signal, through their corresponding horizontal *G-Lines*, the completion of the barrier synchronization to all waiting horizontal slaves. It is worth noting that all participating threads are spinning on a register until the whole process is completed as will be explained in Section 3.2.3. Finally, a detailed explanation of the implementation of these controllers is presented in Section 3.2.4.

3.2.3 Programmability Issues

The *GBarrier* mechanism proposed in this chapter is intended to be used by programmers in a transparent way. For that reason, as shown in Figure 3.5, we propose to provide a special library-level barrier method (*GL_Barrier* in the figure) that encapsulates the functionality of *GBarrier* and that could be used in parallel applications to deal with barrier operations. This barrier method uses a special 1-bit register, called *bar_reg*, to notify the arrival at the barrier by setting its value to one (see the *mov* instruction in Figure 3.5). As explained later in Section 3.2.6, the *bar_reg* register needs as many bits as the number of *GBarriers* provided in hardware (one bit per barrier). In this way, several barrier operations involving different sets of cores (the threads in each set running one application) could

```
GL_Barrier() {
      asm {
            # Arrival at the barrier
              mov 1, bar_reg
            # Wait until all cores arrive
              loop:
                  bnz bar_reg, loop
            # Resume execution
      }
}
```

Figure 3.5: Encapsulating the *GBarrier* functionality into the *GL_Barrier* method.

take place simultaneously. In this way, the register file of each core must be augmented with the *bar_reg* register and the interplay between controllers and these registers must be enabled, switching on the controllers whenever the *bar_reg* registers are written, and resetting the registers and switching off the controllers once all controllers have finished the synchronization (cycle 3 in Figure 3.4). In this way, the synchronization protocol explained in the previous section would be invoked as a result of the activation of the *bar_reg* register by a processor core. Then, each core would enter in a loop waiting until the rest of cores have reached the barrier. Once all cores have set their corresponding *bar_reg* register and the synchronization protocol has been completed, all *bar_reg* registers are reset by the corresponding *GBarrier*'s controllers and then, all cores would leave the loop in order to resume execution.

3.2.4 Implementation of Master and Slave Controllers

In this section, we take a closer look at the implementation of the set of controllers in charge of carrying out the synchronization protocol previously explained.

Figure 3.6 depicts the automata corresponding to each of the four controllers aforementioned: Sh, Mh, Sv and Mv for horizontal slave and master, and vertical slave and master controllers, respectively. Furthermore, over each transition, we also indicate the event that motivates the transition to the next state, and the action that may produce a new event[2] ([EVENT] / [ACTION]). As we can observe, it can be distinguished the following events and actions:

[2]Events that motivate the same action are grouped by using the || symbol.

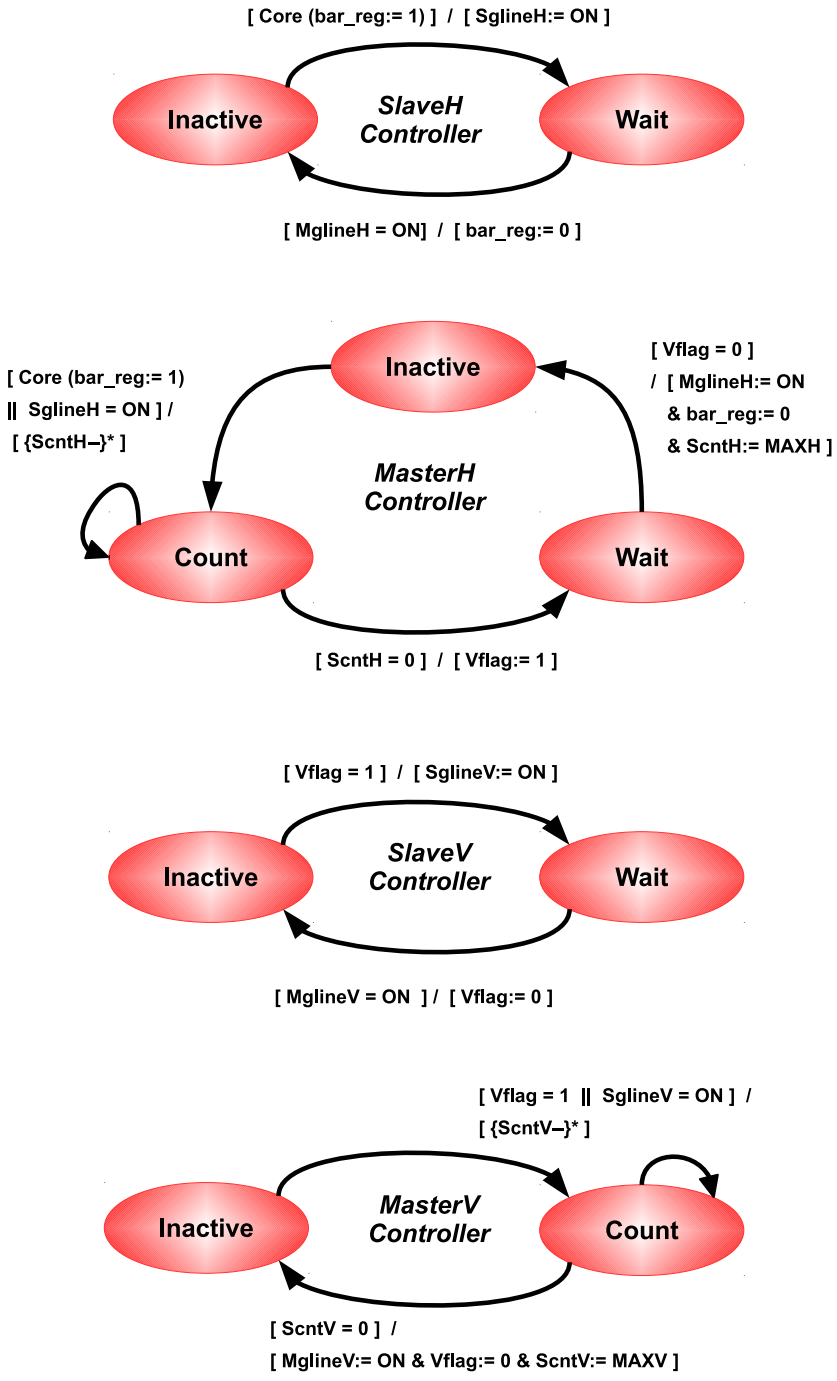


Figure 3.6: Finite state automata that implement the *G-Line* controllers.

- A core writes the *bar_reg* register: *Core(bar_reg:=1)*. Notice that we use := to assign a value and = to compare two values.
- A controller sends or receives a signal through a *G-Line*: *XglineY:=ON*, where X identifies the type of controller (M for master and S for slave), and Y identifies the *G-Line* (V for Vertical and H for Horizontal *G-Lines* respectively). Besides, if X is equal to S, the event may represent the case where several slave controllers send a signal at the same time, thus the use of the *S-CSMA* technique at the corresponding master controller would be employed.
- A master controller updates its *ScntH* or *ScntV* register: *ScntH:=MAXH* or *ScntV:=MAXV*, where, as above explained, the *MAXH* and *MAXV* values correspond to the sum of the number of participant horizontal or vertical slave controllers, plus one if the processor core the corresponding master controller is attached to also participates in the barrier. Note that, the *{ScntH- -}** and *{ScntV- -}** actions for a horizontal or a vertical master controller indicate that the action could be repeated a number of times. For example, if there are three concurrent transmissions from vertical slave controllers (three *SglineV=ON* events), the *ScntV* register has to be decremented three times (i.e. *{ScntV- = 3}*).
- A controller sets or unsets the *Vflag* flag: *Vflag:=1* or *Vflag:=0*, respectively.
- Finally, no event or action: *[]*.

Next, in order to help the reader, we give a comprehensive description of the four automata illustrated in Figure 3.6.

As to the horizontal slave controller (see the SlaveH automaton), every horizontal slave controller remains in the *Inactive* state until the corresponding core arrives at the barrier. Upon arrival, the register *bar_reg* is written, which triggers the event *Core(bar_reg:=1)*, and the horizontal slave controller writes into the horizontal *G-Line* (*SglineH:=ON*). Then, it is switched to the *Wait* state until the horizontal master controller commands to resume execution by writing into its horizontal *G-Line* (the *MglineH=ON* event).

The horizontal master controller (see the MasterH automaton) is switched on when either a horizontal slave controller writes its *G-Line* (the *SglineH:=ON* event) or the local core has arrived at the barrier (*Core(bar_reg:=1)*). In both cases, the horizontal master controller decrements the *ScntH* register (*{ScntH- -}**) and the MasterH automaton transitions to the *Count* state. Besides, in this state the

horizontal master controller continues with the accounting process for every of the signals from the SlaveH controllers by updating the register *ScntH*. Next, once *ScntH* is equal to zero (all horizontal slave controllers in the same row have arrived at the barrier and also the local core), the corresponding *Vflag* is set (*Vflag:=1)*) to notify the corresponding vertical controller. Then, the automaton transitions to the *Wait* state, in which the horizontal master controller remains until the *Vflag* is reset (*Vflag=0*) by the corresponding vertical controller.

The vertical slave controller (see the SlaveV automaton) enters into play when the *Vflag* is set by the corresponding horizontal master controller (*Vflag=1*). This causes the transition to the *Wait* state at the same time that a signal is written into the vertical *G-Line* (the *Sgline:=ON*). Once the vertical master controller notifies that execution can be restarted (*MglineV=ON*), the *Vflag* is reset and the vertical slave controller is switched off.

The vertical master controller (see the MasterV automaton) is switched on when either the *Vflag* flag is set by the corresponding horizontal master controller (*Vflag=1*), or a vertical slave controller has written into its slaves' vertical *G-Line* (*SglineV=ON*). For both events, the register *ScntV* is decremented as explained for the horizontal master controller. Next, once all SlaveV controllers have signaled through the *G-Line* and the *Vflag* is set (i.e. the *ScntV=0* event), the barrier synchronization has been completed and the vertical master controller has to notify the rest of controllers. To do that, the vertical master controller writes into the corresponding vertical *G-Line* (*MglineV:=ON*), sets the *ScntV* to the maximum value and resets the *Vflag* flag. As a result of the *MglineV=ON* event, the vertical slave controllers also reset the *Vflag* flag. Then, the horizontal master controllers signal through the unused *G-Line* the completion of the barrier (*MglineH:=ON*), and also sets the *ScntH* counter to the maximum value. Consequently, the *bar_reg* registers are reset in both horizontal master/slave controllers allowing all cores to resume execution (see Figure 3.5).

3.2.5 Implementation Costs for *GBarrier*

In this section, we discuss the costs that a single *GBarrier* would entail depending on the two kind of technologies evaluated in this thesis: *G-Lines* and *Standard* technologies. As we can see in Table 3.1, we summarize the hardware components of *GBarrier* assuming a 2D-mesh topology for the many-core CMP.

Regarding the *G-Lines* technology, we must deal with the number of *G-Lines* that are used to configure the special network. As already commented on, the *G-Line*-based network deploys separate sets of *G-Lines* per barrier. In particular

Table 3.1: Hardware Cost of *GBarrier* for a 2D-mesh CMP layout with R rows and C columns (R×C = N cores)

***#G-Lines* or #Wires**	$2 \times (R+1)$ or $R \times (C+1)$
Master controllers	$R+1$
Slave controllers	$N-1$
***bar_reg* Registers**	N
***Vflag* flags**	R
***Scnt* Registers**	$R+1$
***bar_reg* size**	1 bit
***Vflag* size**	1 bit
Horizontal *Scnt* size	$\lceil log_2(C) \rceil$ bits
Vertical *Scnt* size	$\lceil log_2(R) \rceil$ bits

each barrier needs a set of $2 \times (R+1)$ *G-Lines*, being R the total number of rows in the CMP. Besides, barrier synchronization is achieved using a set of controllers, which includes $R+1$ master controllers and $N-1$ slave controllers, where N is the total number of cores in the CMP. Each of these controllers would implement the simple synchronization protocol described in Section 3.2.2. In addition, each horizontal master controller requires one flag (the *Vflag* flag) for establishing the local synchronization with its corresponding vertical controller, and also one *ScntH* register. The size of the former depends on the number of *GBarriers* implemented in the system (one bit per barrier). The size of the latter is $\lceil log_2(C) \rceil$ bits for the horizontal master controllers, being C the number of columns in the 2D-mesh topology. Moreover, $\lceil log_2(R) \rceil$ bits are required for the *ScntV* register in the vertical master controller. Finally, the register file in each core must be extended to provide the *bar_reg* register. The total number of bits of the *bar_reg* register will depend on the total number of *GBarriers* implemented in the system (one bit per barrier).

With respect to the *Standard* technology we must take into account that, differently from *G-Lines* technology, this technology implements neither extremely fast links (*G-Lines*) that could be shared by several slave controllers in a particular row/column of the CMP, nor the *S-CSMA* technique, that would enable a master controller to determine how many of its slaves have signaled in a given instant. In particular, every *G-Line* has been implemented by means of a conventional on-chip wire, although we allocate a different wire per slave controller to preclude simultaneous signals from mutual interference. As a result, for every CMP's row

there will be a total of $C-1$ wires for the communications between slaves and master in the gather phase, whilst a single wire per row for the release phase. The same number of wires would be employed for the first column. This results in a total of $R \times (C+1)$ wires ($R \times (C-1+1) + (R-1+1)$) for this implementation of *GBarrier*, as shown in Table 3.1. Additionally, we mimic the *S-CSMA* technique by instructing each master to sample its different slaves' wires in a loop until all expected signals have been received. The remaining hardware costs described for *G-Lines* technology would be the same for this technology.

It is worth noting that, as analyzed in Section 3.3.2, the hardware costs discussed above lead to a marginal area overhead and power dissipation. As we will see, *GBarrier* provides also efficiency because differently from SW-barriers and some implementations of HW-barriers (see Section 3.5), our proposal neither consumes space in memory and local caches with synchronization information nor involves the cache coherence protocol. In this way, *GBarrier* would avoid the significant amount of traffic that those proposals would introduce in the main data network. This would also translate into important energy savings for the interconnection network, as we will show in Section 3.4.3.3.

3.2.6 Generalization of the *GBarrier* mechanism

In this section we explain how the basic proposal discussed until now (hardware support for just one barrier and all participating cores) can be generalized so that it can be successfully implemented in several scenarios.

3.2.6.1 Several *GBarriers* and Subsets of Cores

Up to now, we have described support for just one barrier operation among all cores. However, our mechanism could be easily extended to support a higher number of *GBarriers*. For that, the resources required by one *GBarrier* (and detailed in Section 3.2.5) should be replicated as many times as the number of required *GBarriers*. Since our proposal is specially aimed at carrying out barrier synchronizations efficiently when a significant number of cores is involved, we think that only a very limited number of *GBarriers* would be enough.

On the other hand, we have assumed that all cores in the CMP participate in the barrier. We could also extend *GBarrier* to allow the case when only a subset of them is involved. In this way, the horizontal *ScntH* registers and the vertical *ScntV* register must be initialized with values according to the number of participating threads and their exact location on the CMP. To do that, we would

add one OS-accessible additional register for every *ScntV* or *ScntH* register in order to keep the current number of participating rows and cores for each of them, that is, the *MAXV* and *MAXH* values, respectively. These registers, rather than the number of columns and rows, would be used to set the initial values for the horizontal *ScntH* registers and the vertical *ScntV* register. This initial setup would be done by a system call invoked from a *GL_init()* method called by every participating thread. Note that this operation adds negligible overhead since it must only be performed once. Straight afterwards, the *GBarrier* execution for a subset of cores would be identical to that run by the whole CMP. Even though this design prevents threads from being migrated from one core to another, we strongly believe that this is not a hard constraint given the fact that thread migration is known to be a performance bottleneck. Note that thread migration could be supported by providing an additional mechanism to ensure that the values of the registers can be modified atomically. Finally, it is worth noting that our proposal is also compatible with the use of SW-barrier implementations. This means that even the same application could make use of both *GBarrier* and SW-implementations simultaneously. The latter would be the preferred choice for small subsets of cores because in this case the benefits provided by *GBarrier* would not be significant.

3.2.6.2 Larger Many-Core CMPs

In the *G-Lines*-based implementation technology for *GBarrier*, the master controllers implement an *S-CSMA* technique for counting the number of 1-bit signals transmitted from slave controllers across the *G-Line* they are attached to (see Section 3.2.1). In particular, this technique enables the master controllers to identify up to six different signals transmitted in the same clock cycle. In consequence, every *G-Line* could attach up to one receiver and six transmitters, that is, one master controller and six slave controllers, respectively. In this way, assuming a 2D-mesh physical layout for a CMP, our proposal is restricted to a 7×7-core CMP.

To overcome this limitation, we propose to adopt a hierarchical *GBarrier* design among groups of 7×7 *G-Line*-based networks (*Groups* from now on). For instance, Figure 3.7 illustrates how our proposal could be extended for a 2×7×7-core CMP. As we can observe, there are two *Groups* linked together through two additional *G-Lines*. For clarity, we represent each *Group* as a 2×2-core CMP. Moreover, there are two new controllers attached to these new *G-Lines*: the *Mg* controller, which is an inter-*Group* master controller; and the *Sg* controller, which corresponds

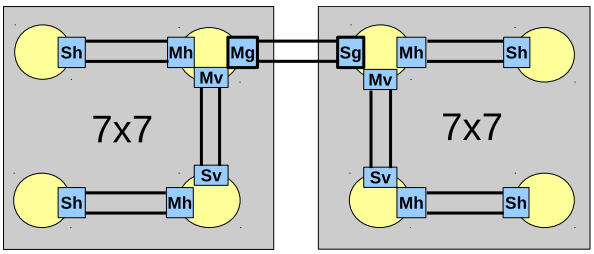


Figure 3.7: Hierarchical *GBarrier* for a 2×7×7-core CMP.

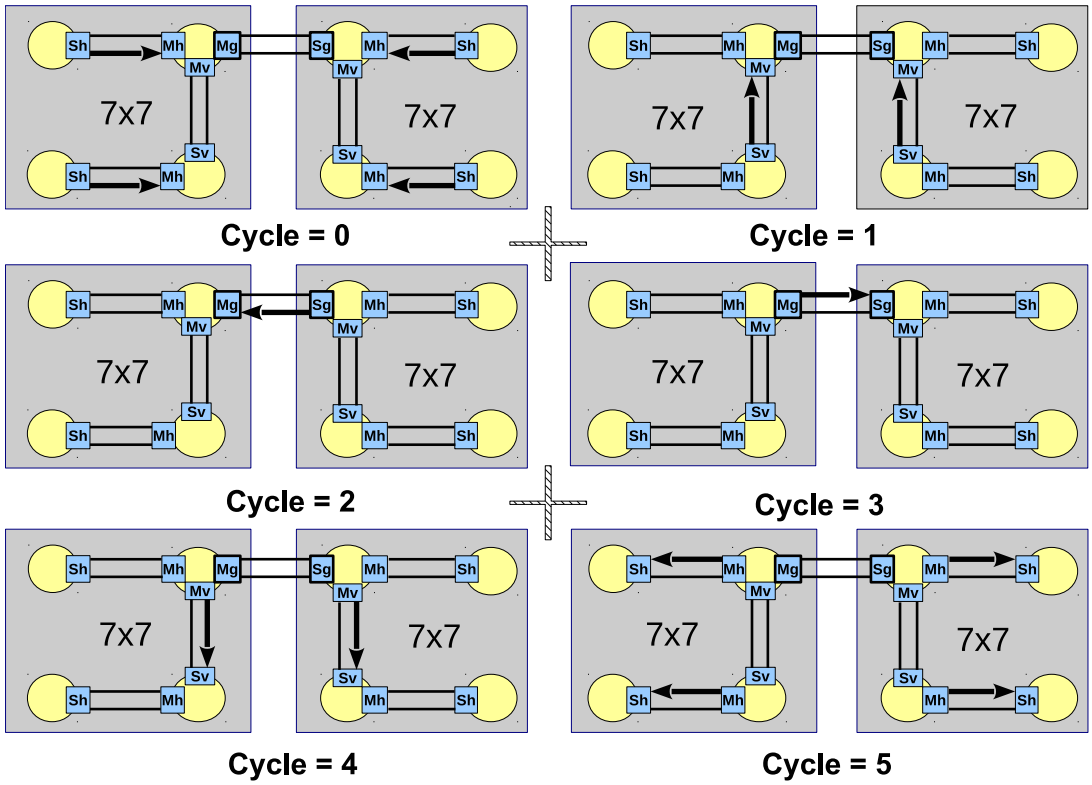


Figure 3.8: Example of barrier synchronization for a 2×7×7-core CMP.

to a inter-*Group* slave controller. The rest of controllers will be referred to as intra-*Group* controllers.

The synchronization protocol for this hierarchical *GBarrier* architecture would operate by following the six steps depicted in Figure 3.8. Note that, we assume the best-case scenario in which all cores execute the same barrier operation at the same time. In consequence, from the figure we can see that each intra-*Group*

controller carries out the same operations as the original *GBarrier* implementation performs (see Figure 3.4). In particular, cycles 0 and 1 for the *account phase*, and cycles 4 and 5 for the *release phase*. Nevertheless, before starting the *release phase*, we have to make sure that each *Group* has completed its *account phase* to proceed. For that, we use the inter-*Group* synchronization between the *Mg* and *Sg* controllers. This synchronization starts by performing the following inter-*Group account phase*. At cycle 2, the *Sg* controller signals, through its *G-Line*, the completion of its intra-*Group account phase*. Once the *Mg* controller has also noticed that its intra-*Group account phase* has already been completed, the inter-*Group release phase* starts. To do so, at cycle 3 the *Mg* controller signals the *Sg* controller by using the unused *G-Line*, and the intra-*Group release phase* can proceed for both *Groups*.

Note that, this new architecture and synchronization protocol do not affect the interface for programmers discussed in Section 3.2.3. Besides, to deal with larger CMPs, we just have to extend the inter-*Group* network to include more *Mg*, *Sg* controllers and *G-Lines* following a similar pattern as employed for intra-*Group* topologies.

With respect to the *Standard* technology, as core count increases in the many-core CMP, there exist new challenges that may restrict the scalability of *GBarrier*. This is due to the fact that efficiency and scalability are non-trivial to materialize when using this technology because of several challenges. First, the RC propagation delay of on-chip interconnects degrades as feature sizes shrink, hence making global wires increasingly slow [34, 94]. Second, propagation delay of logic controllers required by each scheme affects their operating speed, again making relative performance non-trivial. That is the reason why the *Standard* technology is also known as an interconnect-dominated nanoscale technology. To overcome such limitations, we could adopt a similar hierarchical scheme as described for *G-Lines* technology. By doing so, the higher the synchronization stages the shorter the wire lengths will be obtained and then, a lower complexity of master controllers will also be achieved due to sampling fewer slaves. A detailed study of this strategy can be found in [71], where we propose different hierarchical designs to implement hardware barriers in the context of clusterized many-core CMPs such as the HyperCore Processor [120] or Platform 2012 [135].

Alternatively, to also achieve high scalability, our proposal could also be easily implemented assuming the leading-edge nanophotonic technology [123].

3.2.6.3 Simultaneous Multithreaded Processor Cores

The *GBarrier* architecture has been devised to operate on single-threaded processor cores. This section explains how to extend our proposal when simultaneous multithreaded cores (SMT cores) are considered. In this kind of processors, every core's thread has its own register file and shares resources such as the issue window or functional units among others. The problem in SMT cores is that several threads all belonging to the same core would compete for the same *GBarrier*'s resources.

If all executing threads in the processor core belong to the same application, every core's thread would indicate its arrival at the barrier through its private *bar_reg* register (by setting the bit associated with the barrier). Next, to activate the core's *G-Line* controller, a simple AND-gate among all the *bar_reg* values would be used. Once the result of the AND-gate is equal to one, the use of the rest of the *GBarrier*'s resources (*G-Lines*, *ScntH* and *ScntV* counters, and *Vflag* flags) would be the same as for a single-threaded processor core. On the contrary, if the threads executing on the same core belong to different applications, they are forced to use different *GBarriers* for synchronization. In this case, an additional step would be required to selectively enable the AND-gate inputs. Finally, since the number of threads per core is commonly very low, not only due to physical constraints but also to memory footprint constraints, only a few *GBarriers* would suffice to deal with this worst case scenario.

3.3 Performance Implications

In this section, we analyze *GBarrier* to gain insight into its potential impact on performance. First, we start by discussing some considerations taken when using both *G-Lines* and *Standard* technologies to implement our proposal. Next, for both implementations, we show their potential contributions to performance in terms of some important raw statistics such as on-chip area overhead, power dissipation, maximum operating speed and minimum latencies to complete a barrier operation.

3.3.1 Implementation Technologies

To implement *GBarrier*, we have leveraged two different technologies. First, we have made use of the state-of-the-art full-custom *G-Lines* technology explained in Section 1.6. Second, we have employed the standard design methodology,

described in Section 2.2.3, to achieve a cost-effective *GBarrier* implementation at the expense of some negligible degradation in performance, as we will see.

3.3.1.1 *G-Lines* Technology

There were several reasons why we decided to use the *G-Lines* technology to develop our synchronization mechanism for barriers in many-core CMPs. First, the connectivity pattern utilized to deploy the dedicated *GBarrier*'s network (see Section 3.2.1) is based on long 1-bit single-dimension links which perfectly fit into the concept of *G-Lines*. Second, the promising results that could be achieved using this technology in terms of marginal area overhead and minimal power dissipation. Note that, according to the results reported in [142], that show negligible area overhead for a 392-*G-Line* network, the 32-core CMP system evaluated in this chapter (further details in Section 3.4.1) is made up of approximately one-20th of the latter number of *G-Lines*, thereby even lower implications for on-chip area would be obtained. This marginal area overhead will have also a negligible impact on power dissipation. Finally, the *GBarrier*'s synchronization protocol explained in Section 3.2.2 could take advantage of the extremely fast transmissions at 2.5 GHz that the use of the *G-Lines* technology would entail. In this way, we can directly adopt the same theoretical synchronization latencies for the gather and release phases explained in that section.

3.3.1.2 *Standard* Technology

The *GBarrier* architecture has also been implemented relying on the mainstream industrial synthesis toolflow with an STMicroelectronics 45 nm standard cell technology library as that presented in Section 2.2.3. The main reason why we decided to employ this technology was to provide a cost-effective implementation and to precisely quantify the performance losses due to the use of this interconnect-dominated nanoscale technology.

Since RC-based wires are very critical to performance degradation, we have implemented each *GBarrier*'s controller by separating the delay that signals take along the wires, from the effective computation that the controllers require to generate their output signals. Notice that, for small many-core CMPs, the critical path that limits the maximum operating speed in our *GBarrier* infrastructure is defined by the most complex controller (i.e. the master controller that samples signals from the highest number of slaves), but as the wire length increases for larger CMPs, the wires could represent such critical path. Consequently, separating wire delays from controllers delays become essential in order to

Table 3.2: Raw statistics using *G-Lines* and *Standard* technologies for a single *GBarrier* in a 32-core CMP layout.

	Frequency (MHz)	**Latency (cycles)**	**Area (μm^2)**	**Power (mW)**
G-Lines	2,500	4	*Negligible*	26.4
Standard	670	14	5,441	*Negligible*

achieve maximum clock speeds. In this way, by using this technology, we cannot directly assume the synchronization latencies achieved by using *G-Lines*, and a higher number of cycles will be required for the gather and release phases. In addition, to minimize the length of wires, we have situated the master controllers in the central column/row of the 2D-mesh topology, rather than the first column and first row as depicted in Figure 3.2. Note that, in case of *G-Lines* technology this optimization would not be necessary since every *G-Line* is specially designed to implement one-cycle latency, one-bit transmissions across one dimension of the chip.

Finally, for a real characterization of our *GBarrier* proposal, our mechanism has been synthesized by defining non-routable obstructions. Such obstructions are placed to mimic the area of every core of the simulated system explained in Section 3.4.1. In this chapter, we assume that this area is equal to 550×550 μm^2. Additionally, fences are defined to limit the area where the cells of each *GBarrier*'s controller can be placed. Such obstructions and fences also ensure minimum-length routing for the wires in order to reduce their impact on performance and area overhead as the wire length increases.

3.3.2 Raw Performance Statistics

Table 3.2 shows the main raw performance statistics obtained from the use of both technologies to implement *GBarrier*. In particular, we illustrate the maximum operating speed, the latencies of a barrier synchronization and also the area overhead and power that our proposal entails.

As we can see, the maximum operating speed achieved by the *G-Lines* technology is 3.7 times higher than for the *Standard* technology. Moreover, the number of clock cycles employed by the former technology to complete a barrier is 3.5 lower than that achieved by the latter technology. The reason is that every *GBarrier*'s controller and wire involved take a different clock cycle in the synchronization process. Besides, the internal communication using the *Vflag* flag between con-

trollers located in the same core (e.g. Mh and Mv in Figure 3.3) requires an extra clock cycle to achieve the maximum operating speed. Therefore, the superior efficiency of *G-Lines* technology reports roughly a thirteen times faster *GBarrier* implementation.

In addition, negligible overheads in terms of die area are reported for both technologies. First, regarding the *G-Lines* technology, as discussed above, our *GBarrier* infrastructure uses one-20th of the minimal area overhead reported in [142] and then, we can assume that its on-chip area is negligible. And second, for the *Standard* technology, an area overhead for *GBarrier* equal to 5,441 μm^2 is required that corresponds to a negligible 0.06% of the total area employed for the simulated 32-core CMP layout (remember that we assume that each core is 550×550 μm^2 in size).

The latter marginal on-chip overheads will introduce a negligible impact on power dissipation. To exemplify that, we estimate the power dissipated by the *G-Line*-based implementation. To do so, we employ the power dissipation parameters for a 65-nm CMOS process simulated in [142]: 0.6 *mW* per transmitter; 0.4 *mW* per receiver; and 2.4 *mW* per receiver that implement the *S-CSMA* technique. Moreover, according to [142] no static power is dissipated by the *G-Lines*.

To estimate the power dissipation, we must deal with the maximum number of transmitters and receivers in the system operating at once. From the synchronization protocol already explained and illustrated in Figure 3.4, the worst case of power dissipation per clock cycle is when all cores initiate the gather phase at the same time. Therefore, for the simulated 32-core CMP in Section 3.4.1, and considering a 4×8-core 2D-mesh layout[3], there will be seven horizontal slaves per row (i.e. 28 transmitters) signaling the arrival at the barrier, and four horizontal master controllers that count the latter signals through the *S-CSMA* technique (i.e. four receivers). Hence, the total power estimated will be 26.4 *mW* ($28 \times 0.6 + 4 \times 2.4$). Utilizing CACTI [52], the magnitude of this dissipation is less than one-11st of the power dissipated per read port in the L1 caches simulated in this chapter (see Table 3.3).

As a conclusion, the above results suggest that the fastest technology is the most appropriate implementation to materialize *GBarrier*. Although synchronization delay would become the discriminating factor, we have also to consider the major drawback of using *G-Lines*: *The G-Lines technology is a full-custom technology*

[3]For simplicity, we assume that 8 cores per row can be materialized in *G-Lines*. Recall that this technology is limited to 7 cores per row and, for example, a 6×6-core CMP layout must be considered instead to span the simulated 2D-mesh 32-core system.

that is not cost-effective in the embedded computing domain, hence not being within reach of a standard cell design methodology. In consequence, it would be of paramount importance to determine the exact magnitude of such performance degradation when using the *Standard* technology. In case of being negligible, the slower technology would be the preferred *GBarrier* implementation. This experiment will be conducted in Section 3.4.3.1, by comparing synchronization timings of the two *GBarrier* implementations in comparison to the best SW-barrier implementation.

3.4 Evaluation

In this section we give details of our experimental methodology and performance results. We describe the simulation environment and the set of microbenchmarks and scientific applications that we have used in Section 3.4.1. The two SW-barrier implementations the *GBarrier* mechanism is compared with are presented in Section 3.4.2. Finally, the performance results are analyzed in Section 3.4.3 in terms of execution time, network traffic and energy consumption.

3.4.1 Experimental Setup

In order to support *GBarrier*, the Sim-PowerCMP [3] performance simulator presented in Section 2.2.1 has been extended. Remember that, Sim-PowerCMP is a detailed architecture-level power-performance simulator for tiled-CMP architectures that also estimates energy consumption for the full CMP. Table 3.3 summarizes the values of the main configurable parameters assumed in this chapter. As can be seen, we have simulated a 32-core CMP with an aggressive 2D-mesh network built in a 45 nm process technology.

To evaluate the performance benefits derived from *GBarrier*, we have used one synthetic benchmark, three kernels and three scientific applications. First, the synthetic benchmark is intended to measure the latency of barriers themselves. Hence, it helps us provide some insight into the potential benefits that our *GBarrier* mechanism could provide. To do that, we follow the methodology described in [29]: performance is measured as average time per barrier over a 100,000-iterations loop of four consecutive barriers with no work or delays between them. Second, for the kernels we have employed three kernels from Livermore loops [41]. Following the recommendations given in [82], we focus on Kernels 2, 3 and 6. And third, we have considered three scientific applications: Unstructured, EM3D and Ocean. These applications were chosen since

Table 3.3: CMP baseline configuration.

Number of cores	32
Core	3GHz, in-order 2-way model
Cache line size	64 Bytes
L1 I/D-Cache	32KB, 4-way, 2 cycle
L2 Cache (per core)	256KB, 4-way, 12+4 cycles
Memory access time	400 cycles
Network configuration	2D-mesh
Network bandwidth	75 GB/s
Link width	75 bytes

they present a non-negligible fraction of the total execution time due to barrier operations. We would like to point out that all experimental results reported in this chapter are for the parallel phase of all of the benchmarks under study.

We summarize the characteristics of the set of benchmarks used in Table 3.4. For each of them we account for the input size, the total number of barrier executions (*#Barriers*), and the estimated barrier period (the number of cycles on average between two consecutive barrier executions). The latter is calculated by dividing the total number of execution cycles into the total number of barrier executions in every case. Notice that, the barrier period is a simple metric that somehow quantifies the presence of barriers in every benchmark. For example, the Ocean application presents 364 barrier operations every 205,206 cycles on average (see Table 3.4). Consequently, from this high barrier period, we should not expect to obtain a significant fraction of the total execution time due to barriers. The latter result also limits the potential benefits that our *GBarrier* mechanism could provide. A more detailed analysis will be given below in Section 3.4.3.

3.4.2 Barrier implementations

To quantify the benefits of our *GBarrier* mechanism, we consider that barriers found in the benchmarks previously described are implemented by using two SW-barrier implementations: a centralized sense-reversal barrier based on locks (or CSW), and a binary combining-tree or distributed barrier (DSW). On the one hand, in a CSW barrier, each core increments a centralized shared counter when it reaches the barrier, and spins until that counter indicates that all cores are present.

Table 3.4: Configuration of the benchmarks used in this chapter.

Benchmark	**Input Size**	**#Barriers**	**Period**
Synthetic	100,000 iterations	400,000	2,568
Kernel 2	1,024 elements, 1,000 iterations	10,000	3,103
Kernel 3	1,024 elements, 1,000 iterations	1,000	4,953
Kernel 6	1,024 elements, 1,000 iterations	1,022,000	4,908
Unstructured	Mesh.2K, one time step	80	67,361
Ocean	258x258 ocean	364	205,206
EM3D	38,400 nodes, degree 2, 15% remote, 25 steps	198	3,673

On the other hand, in a DSW barrier, there are several shared counters distributed in a binary tree fashion. Thus all cores are divided into groups assigned to each leaf (variable) of the tree. Each core increments its leaf and spins. Once the last one arrives in the group, it continues up the tree to update the parent and so on towards the root. Finally, the release phase is similar but in the opposite direction (towards the leaves).

In general, the implementation of a barrier can be split into three typical stages: the notification stage (S1), when each core indicates its arrival at the barrier; the busy-wait stage (S2), to wait the arrival of the remaining cores; and the release stage (S3), in order to resume execution. At first glance, our *GBarrier* proposal should accelerate all the three stages because they are executed without involving any network transaction or coherence activity. Remember that, our mechanism operates just by means of a simple synchronization protocol implemented atop a dedicated lightweight on-chip network, taking only four cycles (the best-case scenario for a 7×7-core CMP and *G-Lines* technology) to perform a barrier operation among all threads or cores (see Section 3.2.2). However, we could identify two typical situations in which our proposal may entail negligible improvement. The first situation occurs when a parallel application contains a reduced number of barriers or a very high barrier period. This helped us to pick the most significant benchmarks for our evaluation (e.g. choosing Ocean among all of the applications from the SPLASH-2 benchmark suite). The second situation takes place when barrier latency is dominated by the stage S2. For instance, this fact may suggest that the application is under workload imbalance. We will take into consideration these conclusions when analyzing the performance results in the next section.

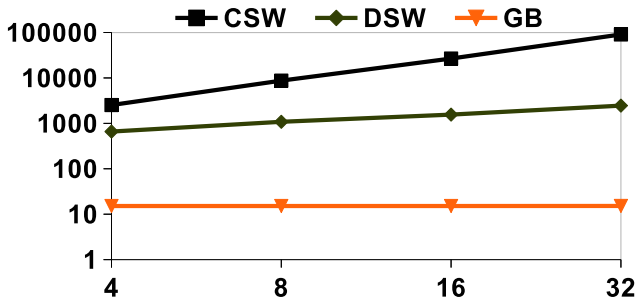


Figure 3.9: Average times for three different barrier mechanisms.

3.4.3 Performance Results

The evaluation of the *GBarrier* mechanism has been carried out taking into account the execution times achieved for the benchmarks shown in Table 3.4, as well as the amount of traffic in the interconnect and the energy-delay^2 product (ED^{2}P) metric for the full CMP.

3.4.3.1 Execution Time

First of all, we consider the implementation of the *GBarrier* that relies on the *G-Lines* technology. Figure 3.9 illustrates the execution times obtained for the synthetic benchmark under study depending on the number of cores in x-axis (from 4 to 32 cores). Notice that y-axis is in logarithmic scale. Remember that, the use of this benchmark allows us to measure the latency of barrier operations themselves. As we can observe, there are three lines depending on the three barrier implementations explained in Section 3.4.2: CSW, DSW and our *GBarrier* mechanism (GB).

From the results presented in Figure 3.9, we can derive two main appreciations. First, the DSW implementation is much more efficient and scalable than the CSW barrier. It is due to the fact that the CSW implementation employs a centralized shared counter among all threads, which clearly becomes a bottleneck as the number of cores increases. In contrast, DSW significantly alleviates contention by using several shared counters distributed in a binary tree fashion. And second, it is clear that *GBarrier* highly outperforms the others in terms of execution time and scalability. On the one hand, the *GBarrier* mechanism drastically reduces execution times of S1, S2 and S3 stages (up to four cycles for the best-case scenario). On the other hand, we deploy a dedicated *G-Line*-based network to implement barrier synchronizations thus removing any coherence activity or

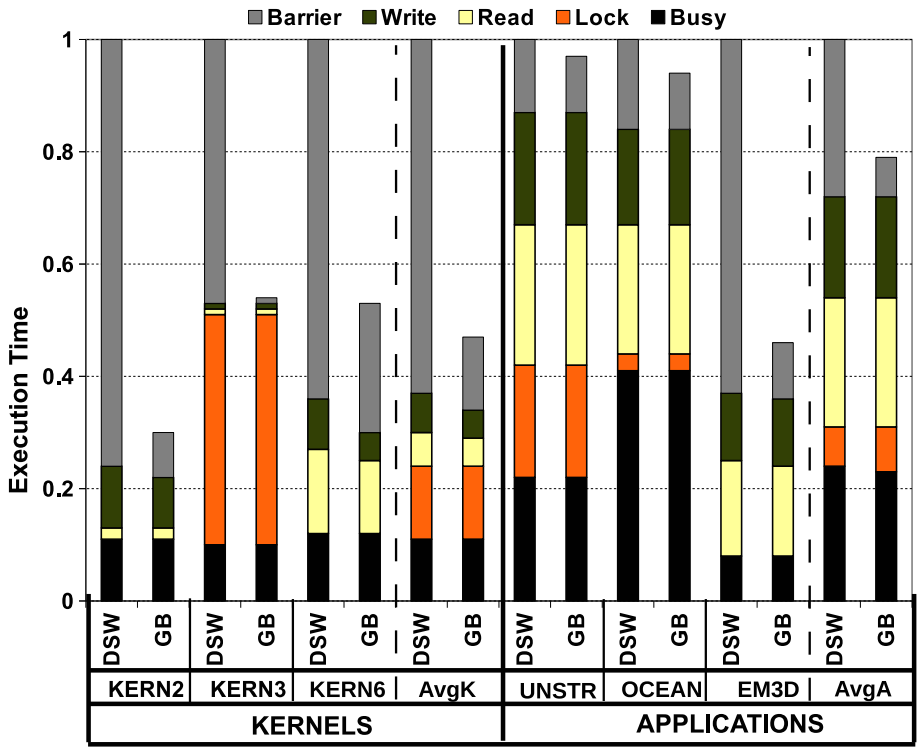


Figure 3.10: Normalized execution time over a 32-core CMP.

synchronization-related traffic in the interconnection network. We would like to point out that our *GBarrier* implementation suffers from a slight overhead in the times obtained (see 13 cycles in Figure 3.9). It is due to the overhead introduced by the simulator when applications call our barrier implementation, because it must be accomplished through its application library.

Figure 3.10 shows the average normalized execution times over a 32-core CMP layout for the rest of applications under study. In particular, for Kernels 2, 3 and 6, and the scientific applications: Unstructured, Ocean and EM3D. Furthermore, we depict the breakdown of execution time depending on the best SW-barrier implementation (DSW) and our hardware barrier mechanism (GB). Execution time is further broken down into several categories: *Barrier* is the time spent on barriers (sum of the time taken in the S1, S2 and S3 stages explained above); *Write* and *Read* are the times spent on memory accesses; *Lock* is the time for lock synchronizations; finally, *Busy* is the time for computational work (e.g. arithmetic operations). In addition, we also illustrate the average times of all kernels and applications for each barrier implementation (see *AvgK* and *AvgA*).

Table 3.5: Speedups for the scientific applications.

Benchmark	Barrier Version	4	8	16	32
UNSTR	DSW	3.32	5.91	10.48	17.43
	GB	3.33	6.01	10.68	17.97
OCEAN	DSW	3.69	7.02	13.46	23.56
	GB	3.70	7.10	13.98	25.06
EM3D	DSW	3.36	5.38	7.32	9.13
	GB	3.42	6.12	10.55	16.82

Regarding the kernels results, we can see that our proposal involves a reduction in execution time of 54% on average (see *AvgK*). In more depth, Kernels 2, 3 and 6 present reductions of 70%, 46% and 47%, respectively. The exact extent of the reduction in each case depends on the barrier period that each kernel has: 3,103, 4,953 and 4,908 cycles, respectively (see Table 3.4). That is, the lower barrier period the higher performance efficiency. For that, Kernel 2 presents the highest reduction in execution time. Moreover, the reductions in execution time obtained also depend on the *Write* and *Read* categories, since our *GBarrier* mechanism operates without involving any memory-related instructions (e.g. see reduction of *Write* category for Kernel 6).

On other hand, the fraction of the execution time that barrier synchronization consumes is lower when scientific applications are considered. In these cases, most of the time is spent on computations and memory accesses (*Busy*, *Write* and *Read* categories), resulting in lower barrier periods. As a result, lower reductions in execution time can be observed for Unstructured, Ocean and EM3D (21% on average). In particular, worse results stem from the applications Unstructured and Ocean since they present a very high barrier period (67,361 and 205,206 cycles, respectively), which translates into reductions of only 3% and 6% in the total execution time, respectively. The exception is EM3D, because it presents significant reductions in execution time (54%) due to its very small barrier period (3,673 cycles).

Table 3.5 shows the speedup results for the scientific applications (Ocean, Unstructured and EM3D) when scaling the number of cores parameter with the values 4, 8, 16 and 32. Moreover, we use two different barrier implementations: DSW in comparison to our *GBarrier* mechanism (GB). From the results shown in Table 3.5, we can extract two important observations. First, all of the benchmarks

Table 3.6: Normalized execution times for *G-Lines* and *Standard* technologies.

	KERN2	**KERN3**	**KERN6**	**UNSTR**	**OCEAN**	**EM3D**
G-Lines	0.30	0.54	0.53	0.97	0.94	0.46
Standard	0.39	0.61	0.56	0.99	0.96	0.55

scale as the number of cores is increased. Second, the exact extent of speedups depends on the efficiency of the barrier implementation we are using. In this way, higher speedups are obtained when employing our *GBarrier* mechanism.

According to the discussion given at the end of Section 3.3.1.2, it would be of paramount importance determining whether the performance losses in terms of synchronization latency derived from the use of the *Standard* technology can be considered negligible. To this end, Table 3.6 shows the normalized execution times with respect to those obtained when the DSW barrier is used, depending on the two kind of *GBarrier* implementations studied in this chapter: *G-Lines*[4] and *Standard* technologies. From the results shown in the table, it can be derived that average performance degradations of 6.3% and 4.3% are reported when using the *Standard* technology for the kernels and scientific applications, respectively. These performance gap is very small if we take into account the significant average reductions in execution time of 48% and 16.6% (kernels and scientific applications) achieved by the *Standard* technology in comparison to the most efficient SW-barrier implementation (DSW). Consequently, we can affirm that our *GBarrier* mechanism is not so dependent on a full-custom technology to provide extremely efficient barrier synchronizations.

Obviously, the performance gap between both technologies will be higher for greater CMP layouts due to the negative effects of the interconnect-dominated nanoscale *Standard* technology. However, the use of a very lightweight interconnection network, that features a hierarchical design, along with a very simple synchronization protocol help relieve such negative effects on performance making the *GBarrier* design really scalable. In particular, in [71], where we explore different hardware-based barrier layouts using *Standard* technology, impressive results are shown for a 64-core CMP layout when comparing performance against the best SW-barrier.

[4]Note that, the results for the implementation that uses *G-Lines* are the same as those presented in Figure 3.10.

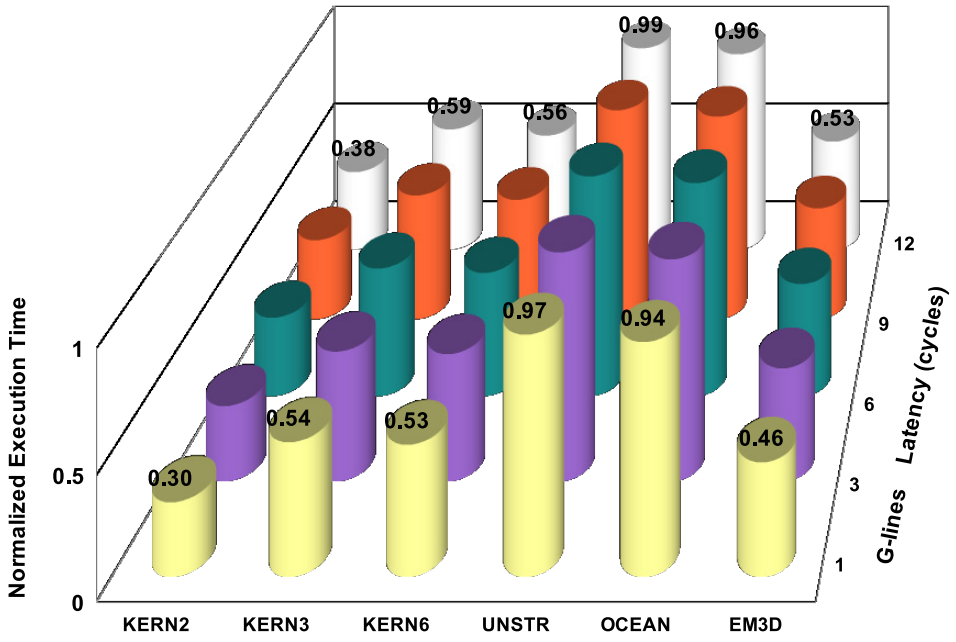


Figure 3.11: Normalized execution times for the benchmarks depending on the latency of the *G-Lines* (a 32-core CMP is assumed).

Finally, we also carried out a sensitivity analysis to evaluate the extent to which our proposal is affected by longer link latencies when considering the *G-Line*-based technology. To do so, we simulate several configurations of the *G-Line*-based network with varying latencies for the links and evaluate the impact that this has on performance. Several clock cycles may be necessary to transmit a signal across one dimension of the chip if, for example, we consider longer links that cannot support a propagation delay of a single clock cycle, or even if lower clock frequencies are required to integrate our *GBarrier* infrastructure in the many-core CMP. Figure 3.11 illustrates the normalized execution times when *G-Lines* take from 1 (results presented in Figure 3.10) to 12 clock cycles (see z-axis in the figure). As we can observe, very small performance losses are derived even when dealing with 12-cycle *G-Lines*. Particularly, performance degradations of just 5.3% and 3.6% on average in the worst case are shown for the kernels and scientific applications, respectively. Note that the results observed for the 12-cycle case are slightly lower than those obtained for the *Standard* technology previously reported in Table 3.6. According to Section 3.3.1, the implementation based

on *Standard* technology is roughly thirteen times slower than the *G-Line*-based infrastructure, so that the former would be roughly equivalent to a 13-cycle *G-Line*-based implementation what explains such similarities.

3.4.3.2 Network Traffic

Our proposal does not generate any coherence messages on the main data network when performing barrier synchronizations. In the end, this translates into significant reductions in terms of network traffic. Figure 3.12 shows the total network traffic across the main data network. In particular, each bar plots the number of bytes transmitted through the interconnection network (the total number of bytes transmitted by all the switches of the interconnect) normalized with respect to the DSW case. Each bar is broken down into three categories: *Coherence* corresponds to the messages generated by the cache coherence protocol (e.g. invalidations and Cache-to-Cache transfers); *Request* comprehends messages generated when load and store instructions miss in cache and must access a remote directory; and finally, *Reply* involves the messages with data.

For the kernels, important reductions in network traffic are achieved (53% on average). In general, these reductions are directly related to the extents of the improvements in execution time previously reported. Moreover, since the simulated L2 cache is shared among the different processing cores, but it is physically distributed between them (see Section 3.4.1), some accesses to the L2 cache will be sent to the local slice while the rest will be serviced by remote slices. This will also affect the timings for lock acquisition and release operations. In contrast, since our *GBarrier* implementation skips the memory hierarchy we have not obtained such negative impact on network traffic. In particular, Kernel 2, 3 and 6 show important reductions of 68%, 37% and 56%, respectively.

Finally, regarding the scientific applications, we can see a slight reduction in network traffic (see 18% in *AvgA*). More specifically, the applications Unstructured, Ocean and EM3D present reductions of 1%, 2% and 51%, respectively. As before, there is a correlation between the fraction of the execution time devoted to barrier synchronization and the amount of network traffic that is saved. In this way, for Unstructured and Ocean we could expect more than 1% or 2% reductions in network traffic, due to the 3% and 5% reduction in execution time, respectively. However, we noticed that the latency of barriers for these benchmarks is dominated by the S2 stage and, as we mentioned, this implies workload imbalance. For the case of DSW, this stage involves negligible network traffic because, once shared variables are loaded in cache, busy-waiting is performed

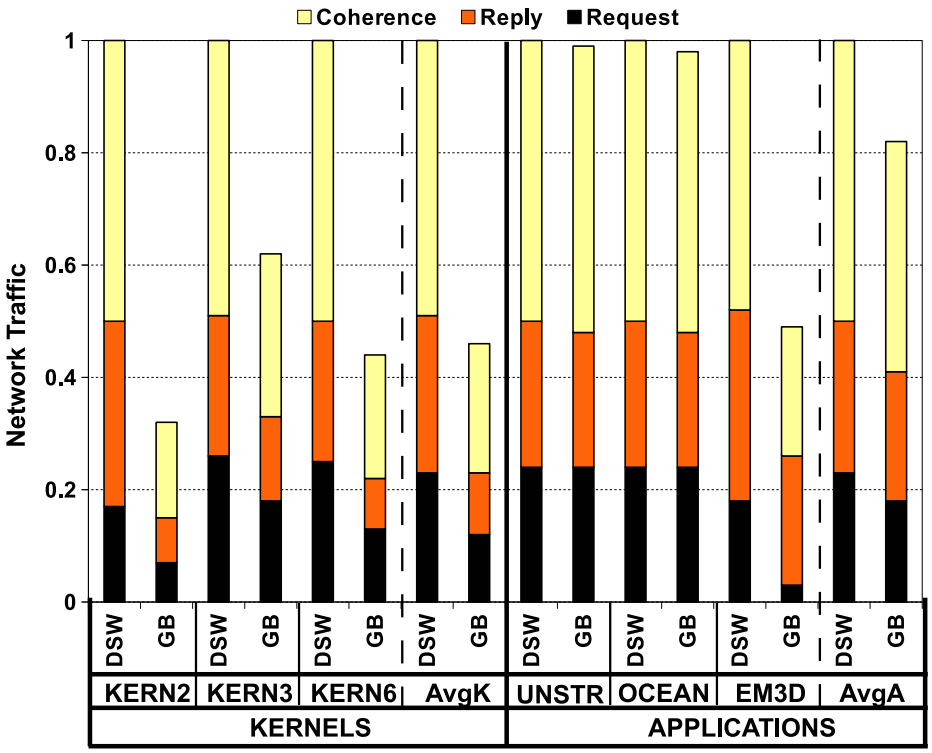


Figure 3.12: Normalized network traffic over a 32-core CMP.

locally. As a result, our *GBarrier* mechanism reports a very low traffic reduction for both benchmarks. Finally, as we expected, EM3D presents a considerable reduction in network traffic (51%) because of its very small barrier period.

3.4.3.3 Energy Efficiency

The use of our *GBarrier* mechanism leads to important reductions in execution time and network traffic, as explained above. In this section, we also quantify the benefits in energy efficiency that our proposal could entail. More specifically, we present in Figure 3.13 the normalized energy-delay^2 product (ED^{2}P) metric for the full CMP. To account for the energy consumed by the *GBarrier* architecture (the *G-Lines*-based network described in Section 3.2.1), we extend the Sim-PowerCMP with the consumption model of *G-Lines* and controllers described in previous Section 3.3.2. As we conclude in that section, the power dissipation associated with our two technology-aware *GBarrier* implementations is negligible, hence the

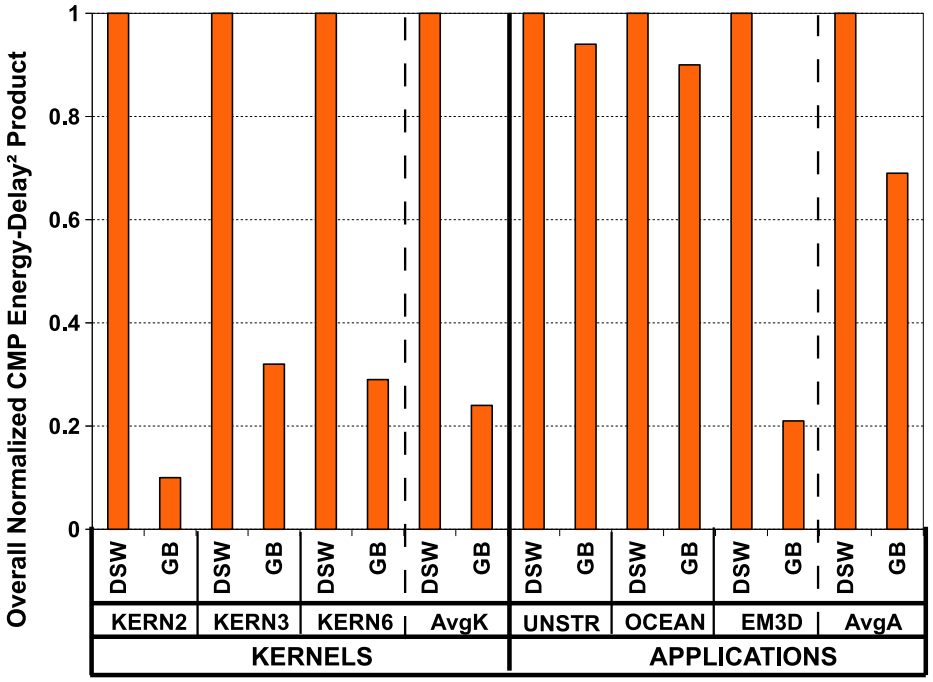


Figure 3.13: Normalized ED^2P metric for the full CMP.

power statistics presented in this section will be mainly due to the improvements in execution time and network traffic reported in previous sections.

As in the previous two sections, all results in Figure 3.13 have been normalized with respect to the DSW case. As can be observed, important improvements in the ED^2P metric of the whole CMP are achieved when applying our proposal. In particular, the *GBarrier* mechanism brings average improvements in ED^2P of 76% and 31% for the kernels and scientific applications, respectively. Particularly, the Kernel 2, 3 and 6 show reductions of 90%, 68% and 71%, respectively. Additionally, reductions of 6%, 10% and 79% are achieved for Unstructured, Ocean and EM3D.

In general, the magnitude of these savings is directly related to the extents of the improvements in execution time and network traffic previously reported. We have found that when *GBarrier* is employed, the number of instructions executed per barrier operation is drastically reduced. Note that while DSW barrier must deal with a distributed shared counter in a binary tree fashion, *GBarrier* only needs a single assignment instruction on a register to notify the arrival at the

barrier (see Section 3.2.3). Obviously, less instructions executed means less energy consumed in the processor cores.

Moreover, since we reduce the latency to notify the arrival at the barrier (S1 stage), the busy-wait process (S2 stage) is also shortened with *GBarrier*. While busy-waiting, a processor core repeatedly accesses the L1 cache to check the value of a shared variable. In this way, shorter busy-waiting implies less accesses to the L1 cache, and therefore, less energy consumed in this structure. Finally, given the fact that our proposal skips the memory hierarchy, we save all the energy derived from coherence activity when barriers are executed. In particular, we remove all of the L1 cache misses related to barrier operations and the corresponding messages transferred across the interconnect. This brings reductions in the energy consumed at the L2 cache banks and the interconnection network.

3.5 Related Work

To overcome the performance limitations imposed by SW-barriers, there have been proposed several hardware-based optimizations in the context of both traditional multiprocessors and, more recently, CMPs. In this section, we make an attempt at categorizing most of them in terms of the part of the system they improve or augment: memory-based approaches, network-based approaches and global lines approaches.

Regarding memory-based approaches, Goodman et al. [81] proposed a set of efficient primitives for process synchronization based on the use of synchronization bits (syncbits). Syncbits are logically associated with every block in memory to provide a simple mechanism for mutual exclusion. The T3E multiprocessor [131] augments the memory interface of the DEC 21164 microprocessor with a set of explicitly-managed external registers (E-registers). All remote communication and synchronization is done between these registers and memory. Moreover, a set of 32 synchronization units (BSUs), accessible as memory-mapped registers, are provided per processor to perform barrier/eureka synchronization. More recently, Sampson et al. [82] presented barrier filters, a mechanism to implement fast barrier synchronization on CMPs. The key idea is that they ensure that all threads arriving at a barrier require an unavailable cache line to proceed. Then, the barrier filter starves their requests until they all have arrived. Monchiero et al. [100] proposed a hardware module to optimize busy-waiting synchronization in CMPs. This module is integrated in the memory controller, namely the Synchronization-operation Buffer (SB). The SB manages locally the

polling on shared variables, avoiding traffic in the network and memory accesses. Subsequently, Zhu et al. [152] proposed a small buffer attached to the memory controller of each memory bank, called the Synchronization State Buffer (SSB). This buffer provides an illusion that the entire memory is tagged at word-level like in a *full/empty* bits based system. To do this, the SSB records and manages the states of frequently synchronized data.

Differently from these previous approaches, our proposal decouples completely barrier synchronization from any kind of memory-related activity.

Regarding network-based approaches, Hsu and Yew [151] proposed a multi-stage shuffle-exchange network to efficiently handle synchronization traffic of SW-barriers by combining packets in the switches in order to relieve hot-spot congestion from the network. Olnowich [50] presented an efficient technique for handling SW-barriers by using a special hardware at the network adapter level. This architecture enables all processors to both drive and receive data at the same time such as multi-drop bus and broadcast communications. Other implementations are based on including a dedicated interconnection network to carry out synchronizations. For example, the network architecture of the Connection Machine CM-5 [21] contains a dedicated network (control network) to perform synchronizations of an entire set of processors through specific messages interchanged between outgoing and incoming FIFO queues at the network interface level. In addition, the Blue Gene/L [116] also contains a dedicated interconnection network for barrier synchronization. Sartori and Kumar pointed in [83] that although a dedicated interconnection network manages barrier operations efficiently, its integration in future many-core CMPs may not be a feasible solution due to the large on-chip area and power dissipation that it could entail. They propose three barrier implementations, that are hybrid of software and hardware aimed at achieving closer approximation to the performance of a dedicated interconnection network but at a fraction of the cost.

Differently from any of the above proposals, *GBarrier* operates independently of the main data network, thus removing all synchronization-related traffic. Moreover, we use a very reduced number of state-of-the-art global links that introduce negligible area overhead and power dissipation.

Finally, regarding global lines approaches, the Sequent Balance system [12] uses chips attached to processors to provide support for interrupt distribution, low-level mutual-exclusion, and configuration and error control (the System Link and Interrupt Controllers, or SLICs). For that, two specific global lines from the system bus are used to communicate SLICs by means of commands from a simple message-passing protocol, not affecting bus bandwidth. SLIC commands

implement *test&set* instructions which also give support to classic higher-level synchronization primitives such as locks and barriers. On the contrary, our method does not need any command to perform synchronizations, which are performed by means of signals transmitted through special global lines. Moreover, our hardware approach is much more simpler than SLICs are, and it is integrated in the context of many-core CMPs. Beckmann and Polychronopoulos [23] presented a hardware scheme specifically designed for fast barrier synchronization in the context of large-scale multiprocessors. This architecture is scalable and supports a large number of concurrent barriers by replicating hardware resources. In more detail, the actual barrier is a single-bit register BR which is visible to all processors through special lines. When BR is set to 1, processors are blocked at the barrier. When BR is set to 0, the barrier is clear and processors may proceed to execute. In case of a multiprocessor with P processors, there is also a P-bit wide R register associated with the barrier register BR. Then, each processor has its own bit from R which is set to 0 in case of arrival at the barrier. The R register is connected to a zero-detect logic, which determines when all bits of R are 0 (i.e. all processors are waiting at the barrier). In latter case, BR is set to 0 and all processors resume execution. Shang and Hwang [132] presented a distributed and hardwired barrier architecture for fast synchronization in cluster-structured multiprocessors. Moreover, they develop a set of synchronization primitives for explicit use of distributed barriers dynamically. To do so, they use a distributed wired-NOR architecture to detect the asynchronous arrivals of different processes at the barrier. Makhaniok and Männer [99] proposed a method to synchronize massively parallel processes in distributed multiprocessor systems. This scheme is based on a synchronizer that uses three bus lines P, Q and R. Each synchronization unit is connected to these lines and can assert onto them its individual binary signals p, q and r. Thus every line carries the wired-OR of the signals asserted by the synchronization units, and all synchronization units read back this value. This involves a synchronization protocol which is composed of different steps depending on the different values of lines P, Q and R.

In contrast, our *GBarrier* infrastructure has been specifically devised for many-core CMPs, relies on very lightweight on-chip network and a very simple synchronization protocol, which features extremely faster communications minimizing considerably synchronization latency. Moreover, our mechanism does not entail any wired-OR or wired-NOR logic to detect the arrivals at the barrier and a more scalable and distributed accounting is employed, because multiple arrivals can be detected during the same clock cycle. Recall that, every row of the CMP has a

master controller that independently receives and counts the signals transmitted from its corresponding slaves.

In the context of CMP architectures, Krishnan and Torrellas [146] proposed a hardware mechanism to support communication and synchronization of registers between on-chip processors for an efficient consumer/producer model. Their proposal is based on a Synchronized Scoreboard (SS) that is provided per processor. The SSs are connected with a broadcast bus on which register values are transferred. In contrast, rather than employing a bus, our architecture employs a more scalable communication network using dedicated state-of-the-art point-to-point links among the participant cores. Besides, instead of register values, only 1-bit transmissions are needed by our synchronization protocol leading to energy and area savings. Additionally, Cyclops [20] is a highly parallel processor-and-memory system on a chip (32-quad-core CMP architecture). This architecture implements a fast barrier operation through a special purpose register (SPR). It is actually implemented as a wired OR for all the threads on the chip. Each thread writes its SPR independently, and it reads the ORed values of all the threads' SPRs. The register has eight bits which provides four distinct barriers (two bits per barrier). One of the bits holds the state of the *current* barrier cycle whilst the other holds the state of the *next* barrier cycle. All threads participating in the barrier initially set their *current* barrier bit to 1. The threads not participating in the barrier leave both bits set to 0. Then, when a thread reaches the barrier it writes 0 to the *current* bit, thereby removing its contribution to the *current* barrier cycle, and one to the *next* bit. Hence, the barrier is completed when all *current* bits become 0. Furthermore, the use of the *current* and *next* bits are interchanged after each execution of the barrier. To communicate the SPRs' values, Cyclops employs a dedicated 16-bit bus which enables the completion of a barrier operation among all threads in only a few dozens of cycles [156].

In contrast, rather than buses, our proposal communicates signals through a more scalable on-chip network based on 1-bit width links deployed in a hierarchical layout. Moreover, as aforementioned, our accounting process is distributed, hence more scalable.

Finally, TLSync [79] is a sophisticated design for barrier synchronization that provides very efficient barriers although being fully dependent on non-standard technology, namely *Transmission Lines* [15]. In particular, the process of synchronization is performed by allocating different radio-frequency bands from the high-frequency part of the spectrum per barrier, thereby allowing multiple groups of threads to be concurrently synchronized very quickly. While this is a very efficient hardware design, a successful implementation is restricted to

leading-edge technology thus not being within reach of a standard cell design methodology. In contrast, in light of the impressive performance results shown in this chapter, a cost-effective implementation of *GBarrier* is also feasible.

3.6 Conclusions

Developing efficient barrier synchronization is recognized a big challenge as the core count increases in future many-core CMPs. In particular, SW-barriers do not keep up with computational power that features this kind of systems, thereby limiting efficiency and scalability. In this chapter we propose *GBarrier*, a novel hardware-based barrier mechanism specifically designed to enable efficient barrier synchronizations in the context of future many-core CMPs.

The *GBarrier* mechanism here developed consists of two main components: First, a very lightweight dedicated on-chip network that could be deployed in a hierarchical layout for scalability. The second is a simple synchronization protocol that, while performing a barrier synchronization, coordinates the actions of the controllers attached to the links of the *GBarrier*'s network. Differently to traditional approaches based on the use of atomic read-modify-write instructions operating on shared-memory positions, our proposal does not have any influence on the memory system. In this way, we avoid all coherence activity and barrier-related network traffic that traditional approaches introduce and that restrict scalability. Additionally, we have discussed how *GBarrier* can be easily adapted to different scenarios and system configurations. In particular, we have extended our infrastructure in order to support: several *GBarriers*, group of cores, larger many-core CMPs and SMT processor cores.

We have evaluated two implementations of the *GBarrier* mechanism. The first made use of state-of-the-art full-custom technology, namely *G-Lines*, whilst the second is based on a mainstream industrial toolflow and standard cells. While the on-chip area overhead and energy consumed by both implementations can be considered negligible, the former technology has been used to report minimum synchronization latency, whereas the latter leads to a cost-effective implementation because it is within reach of a standard cell design methodology.

We integrate both *GBarrier* implementations into a detailed execution-driven simulator of a 32-core CMP running a set of benchmarks (which includes a microbenchmark, three kernels from Livermore loops and three scientific applications), in order to quantify the benefits that our proposal entails. Moreover, we compare performance against one of the most efficient SW-barrier imple-

mentation (a tree barrier) in terms of execution time, network traffic level and energy efficiency. From this study, both *GBarrier* implementations report very similar reductions in execution time, thus not making our proposal so dependent on a full-custom technology to achieve extremely efficient synchronization in many-core CMPs. In particular, for the kernels and the scientific applications under study our proposal brings average reductions of 54% and 21% in total execution time, resulting in improved scalability for the applications. The fact that our proposal does not rely on shared memory positions and the cache coherence protocol saves a significant amount of messages on the main interconnection network (reductions of 53% and 18% in network traffic are observed for the kernels and applications, respectively). Finally, all these gains lead to improvements of 76% and 31% in the energy-delay^2 product (ED^{2}P) metric for the full CMP, for the kernels and scientific applications respectively.

Chapter 4

GLock: An Efficient Infrastructure for Highly-Contended Locks

4.1 Introduction and Motivation

Lock synchronization in parallel applications has long been devised to ensure that a block of code manipulating a shared data structure, namely critical section (CS), is executed by only one process or thread at a time (i.e. the lock owner), thereby guaranteeing mutual exclusion among processes or threads and preserving the integrity of the shared data [40].

In shared-memory parallel systems, this kind of synchronization mechanism commonly comprises a pair of operations. First, the *lock* operation that a thread utilizes before executing the CS to request the lock ownership. And second, once the thread becomes the lock owner and executes the CS, the *unlock* operation, that is executed straight afterwards the CS in order to release the lock ownership, so that another thread can become the next lock owner.

Typical software-based implementations for *lock/unlock* rely on a combination of memory operations on shared variables that involve special instructions such as *LL/SC*, or atomic read-modify-write instructions like *test&set*. Nonetheless, the use of shared variables for lock synchronization has two important implications for performance and scalability, especially in future many-core CMPs. First, the cache coherence protocol must come into play in order to maintain the consistency of shared variables across all levels of the memory hierarchy. Coherence activity translates into traffic injection in the interconnection network. As a result, an

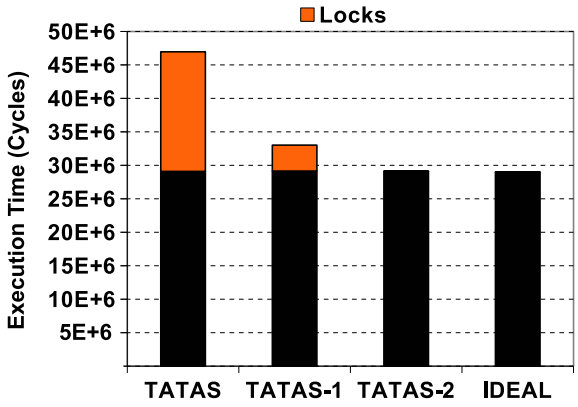


Figure 4.1: Potential benefits for Raytrace when using *ideal locks*.

ever-growing amount of resources may need to be devoted to support lock synchronization as the core count increases. Moreover, lock acquisition and release operations timing is deeply affected by the performance and scalability of the cache coherence protocol especially under the presence of highly-contended locks. Second, lock contention has long been recognized as a key impediment to performance and scalability since it causes serialization [110]. Consequently, the longer the idle time spent on lock acquisition and release operations, the larger the parallel efficiency reduction.

As an evidence, we show in Figure 4.1 the potential benefits to performance when lock synchronizations do not involve the cache coherence protocol and have zero latency. To do so, the Raytrace application from the SPLASH-2 benchmark suite [128] is run by using distinct lock implementations (for details of the evaluation see Section 4.4). In each case, we highlight in orange color the fraction of the execution time due to the locks. Shared-memory-based locks use `test-and-test&set` (see TATAS bar in Figure 4.1). In turn, ideal locks (see IDEAL bar in Figure 4.1) do not deal with the cache coherence protocol to eliminate any inherited performance or scalability side-effects. Besides, lock acquisition and release operations take a single clock cycle each to minimize serialization due to contention. As expected, ideal locks clearly outperform shared-memory-based locks since the lock acquisition and release operations account for a significant fraction of the execution time in Raytrace. However, a post-mortem analysis of Raytrace lock usage reveals that only 2 out of its 34 locks are highly-contended. In this sense, if all the locks other than the highly-contended ones are implemented using regular shared-memory-based locks, a reduction in the execution time

similar to that of ideal locks is obtained (see TATAS-1 and TATAS-2 bars[1] in Figure 4.1). The latter result suggests that only highly-contended locks can truly benefit from a more efficient lock implementation.

In this chapter, we present and evaluate a new lock synchronization mechanism aimed at accelerating highly-contended locks. Our proposal, namely *GLock*, is a lightweight on-chip network infrastructure devoted to implement a very simple token-based message-passing protocol providing extremely efficient execution for highly-contended locks. As with the *GBarrier* mechanism presented in Chapter 3, we have explored two different technologies to implement *GLock*. On the one hand, we make use of the state-of-the-art full-custom *G-Lines* technology introduced in Section 1.6, that enables almost speed-of-light 1-bit communications across one dimension of the entire chip. On the other hand, we employ the mainstream industrial toolflow with standard cells in an advanced 45 nm technology presented in Section 2.2.3, in order to obtain a cost-effective implementation for our proposal at the expense of some negligible performance loss.

To show the benefits derived from *GLock*, we integrate both *GLock* implementations into a 32-core CMP performance simulator [3], and we compare their performance results against the most efficient software-based implementation for highly-contended locks considered to date (*MCS Locks*). To do so, we use several microbenchmarks and real applications from the SPLASH-2 benchmark suite [128]. In particular, our *GLock* mechanism reports average reductions of 42% and 14% in execution time for real applications and microbenchmarks, respectively. The synchronization improvement that brings *G-Lines* with respect to the other slower standard technology implementation is again very small in comparison to the much higher execution times reported by the most efficient software-based implementation. In consequence, a negligible penalization of 1.6% and 1.3% on average for the latter reductions in execution time has been found, respectively. Moreover, given the fact that *GLock* does not deal with the main data network, average reductions of 76% and 23% in network traffic are obtained. These traffic reductions also lead to average savings of 78% and 28% in the energy-delay^2 product (ED^{2}P) metric for the full CMP, respectively. Finally, we have also evaluated the area overhead and power dissipation that each technology-aware *GLock* implementation would entail, concluding that both of them are negligible regardless of the technology employed.

[1]TATAS-X means that one (X=1) or two (X=2) of the highly-contended locks have been implemented as ideal locks.

The rest of the chapter is organized as follows. We present our *GLock* mechanism in Section 4.2. Next, in Section 4.3 we discuss some important performance implications when using our proposal. Section 4.4 describes our simulation environment and analyzes the obtained performance benefits in terms of reductions in execution time, network traffic, power dissipation and on-chip area overhead. Moreover, the related works about efficient lock synchronization mechanisms are discussed in Section 4.5. Finally, Section 4.6 presents our main conclusions.

4.2 The *GLock* Synchronization Mechanism

In this section, we present our proposal to build an efficient synchronization mechanism for highly-contended locks in many-core CMPs. To do so, we will focus on describing the hardware components required and the synchronization protocol employed, rather than going into any technical aspects of the two implementation technologies used (further details in Section 4.3.1). In more depth, we start by describing the dedicated on-chip network that our proposal entails. As a case study, we choose a CMP with a 2D-mesh data interconnection network with R rows of C cores each (for a total of $N = R \times C$ cores), although our proposal is not restricted to this topology. Next, we show how the *GLock* mechanism would operate. After that, we describe the interface for programmers and provide details about the implementation of the set of controllers required by our proposal. Finally, we analyze the hardware resources required by *GLock* and propose how our mechanism can be generalized to operate in several scenarios.

4.2.1 Dedicated On-Chip Network Architecture

The *GLock* mechanism proposed in this chapter relies on a dedicated on-chip network as can be observed in the example in Figure 4.2. For simplicity, we concentrate on a version of the proposed network providing support for one lock. As we can see, the network is made up of two kind of components. *Links* (horizontal and vertical finer black lines), that are used to transmit the signals required by the synchronization protocol; and controllers (R, Sx and Cx), that actually implement the synchronization protocol.

Every *link* is simply a wire that enables the transmission of one bit of information across one dimension of the chip, employing one *link* per transmitter and lock. Every *link* will be used to request the associated lock and grant lock acquisitions. In this way, for any 2D-mesh layout the total number of *links* per

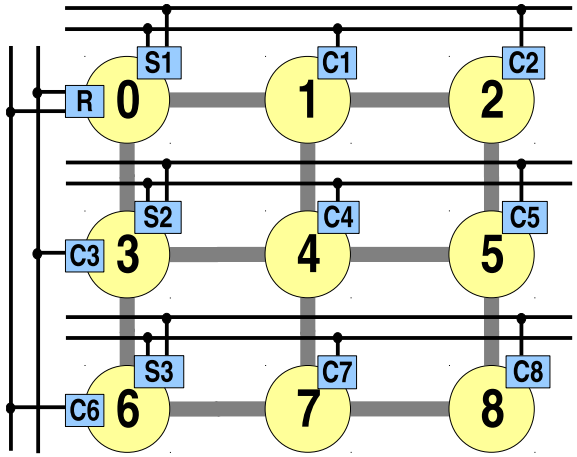


Figure 4.2: *GLock* architecture for a 9-core CMP with a 2D-mesh network.

lock that would be needed is equal to $N-1$, where N is the number of cores of the CMP (e.g. eight *links* for the 9-core CMP shown in Figure 4.2). It is worth noting that our proposal is aimed at providing this kind of hardware support just for a very limited number of locks, enabling the opportunity to deal with very efficient highly-contended lock synchronizations with marginal area overhead (see Section 4.3.1).

In addition to the *links*, our proposal also incorporates a set of controllers. In particular, we distinguish two types of controllers: the *local controllers* (Cx in Figure 4.2) and the *lock managers* (R and Sx in Figure 4.2). The *local controllers* send and receive signals to and from their corresponding *lock managers* through their dedicated *links* (e.g. $C1$ sends and receives signals to/from $S1$). The exception is when the *local controller* is located in the same core as its associated *lock manager*. In this case, the functionality of the *local controller* is encapsulated in the *lock manager*, and communication is performed locally by means of a flag. For example, $S1$ monitors not only signals from *local controllers* one and two ($C1$, $C2$) through their corresponding *links*, but also from the local core through an internal flag (for clarity, this flag is not shown in Figure 4.2).

The *lock managers* control lock ownership by monitoring signals from either *links* (remote cores) or the flags (local core). Besides, *lock managers* are divided into two groups: primary and secondary *lock managers*. Secondary *lock managers* (Sx) are responsible for monitoring signals from their corresponding *local controllers*, whereas the primary *lock manager* (R) is responsible for monitoring signals from

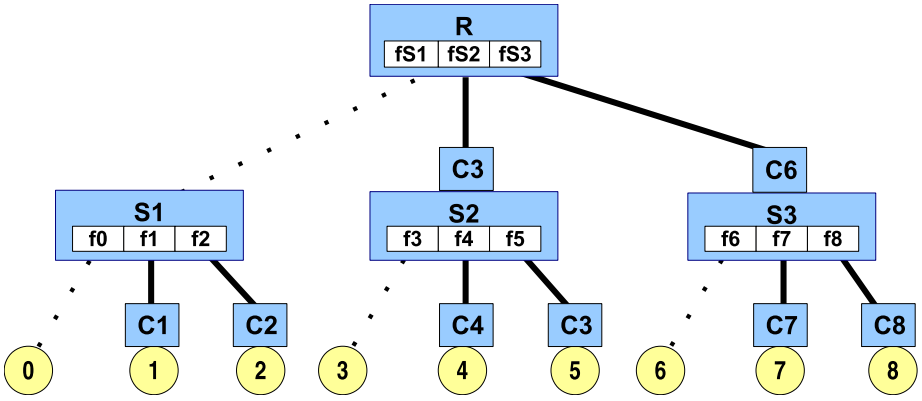


Figure 4.3: Logical view of the *link*-based network for a 9-core CMP with a 2D-mesh network.

the secondary ones. Primary and secondary *lock managers* communicate with each other by means of the vertical *links* shown in Figure 4.2.

Finally, to have a clear understanding of our proposal, we represent the architecture described above as the hierarchy shown in Figure 4.3. In particular, the dedicated network that our proposal is based on can be represented as a three-level hierarchy. The root of the hierarchy is the primary *lock manager*. The secondary *lock managers* would be located at the intermediate nodes. Finally, the leaves of the hierarchy would be the processor cores (with the *local controllers*). All elements are connected using *links* (continuous lines) or locally by means of an internal flag (dashed lines). The flags (fx and fSx) store the signals sent by the controllers to the corresponding *lock manager* (primary and secondary). In this way, we need flags not only to store the signals sent between Sx and the *local controllers* (one flag per Cx controller: $f1$ for $C1$, $f2$ for $C2$, etc.), but also to store the signals transmitted between R and Sx (one flag per Sx controller: $fS1$ for $S1$, $fS2$ for $S2$, etc.).

4.2.2 Synchronization Protocol

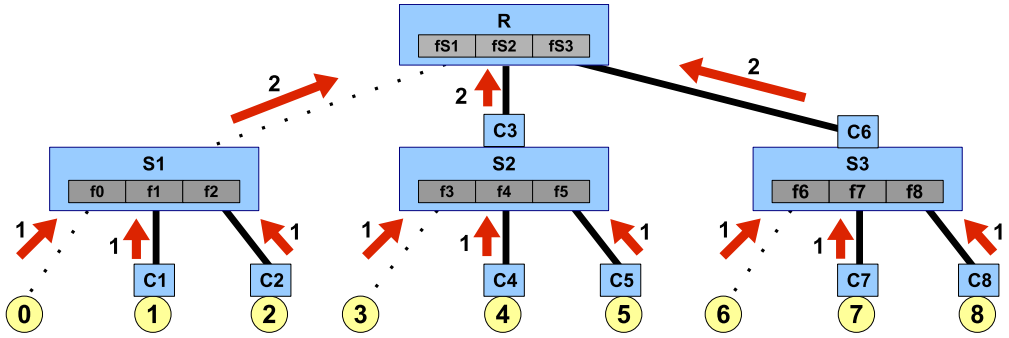
The synchronization protocol implemented on top of the network previously described is based on the exchange of 1-bit messages (signals) between the *local controllers* and the *lock managers*. More specifically, the protocol uses three types of signals to perform a lock synchronization. The REQ and REL signals, which are sent from the *local controllers* to their corresponding *lock manager* to ask for the lock and to release the lock, respectively; and the TOKEN signal which is

sent from a *lock manager* to a particular *local controller* to grant access to a lock. In addition, these signals are also transmitted between primary and secondary *lock managers* in a lock synchronization. In particular, the secondary *lock managers* ask for the lock by sending the REQ signal to the primary *lock manager* and receive authorization from the latter through the TOKEN signal. Similarly, after the lock is released, a secondary *lock manager* notifies the primary one by means of the REL signal.

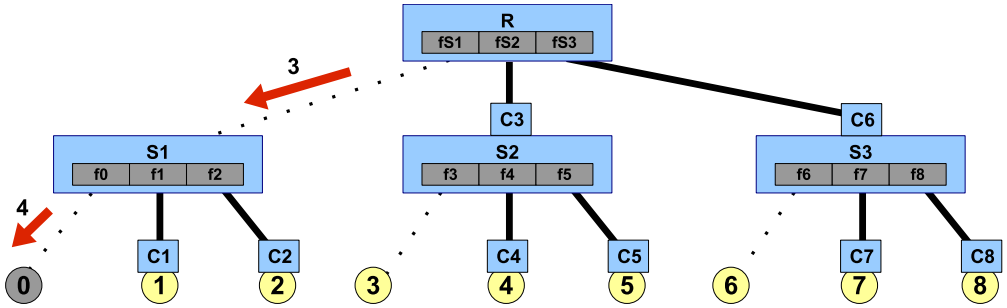
Lock managers (both the primary and secondary ones) use a round-robin strategy to grant the lock among those processor cores which are competing for becoming the next owner. Let's assume that all of the cores in Figure 4.3 send the REQ signal to their corresponding secondary *lock manager* at the same time. In this case, the TOKEN signal granting the lock would be received by Core0 first; then, once Core0 has released the lock, Core1 would become the next holder; and so on, until Core8 is reached. Next, the process would start again from Core0 if there are additional pending lock requests. Since the *GLock* mechanism is aimed at accelerating highly-contended locks we do not expect that the election of the strategy to grant the lock in these situations will have any impact on performance. However, this is a key design point to ensure the fairness expected from a lock implementation [29]. The latter is the reason why we use the round-robin strategy.

As an example of how the synchronization protocol works, Figure 4.4 presents the case where the nine cores of the CMP depicted in Figure 4.2 try to get access to the lock at the same time. To clarify the explanation, the arrows in the figure mark the sense of the transmissions. Moreover, each arrow is labeled with the cycle in which communication occurs, starting with cycle 1. It is worth noting that we are assuming theoretical synchronization latencies that may not be reflected in the exact number of clock cycles required for the two physical *GLock* implementations (see Section 4.3.1). Finally, we highlight with dark gray the flags that are written and the core that acquires the lock in each case.

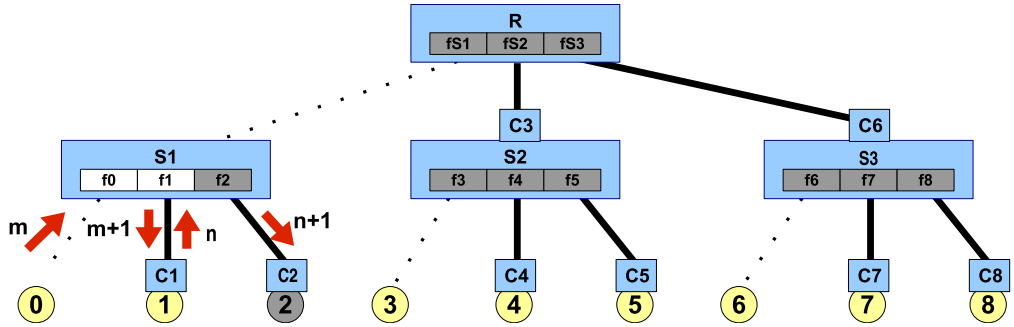
At cycle 0, all cores try to get the lock (see Figure 4.4a). To do so, every *local controller* (Cx in the figure) sends the REQ signal at cycle 1 to the corresponding secondary *lock manager* (Sx in the figure). As a result, all fx flags would be written, and each Cx would be busy-waiting until the TOKEN signal is received. At cycle 2, once each Sx detects that at least one of its fi flags has been written, REQ signals towards the primary *lock manager* (R in the figure) are sent in order to write the corresponding fSx flags. At this moment, R must make a decision about the secondary *lock manager* that will be granted the lock ownership. This process is shown in Figure 4.4b. In this case, R would choose $S1$ by following the



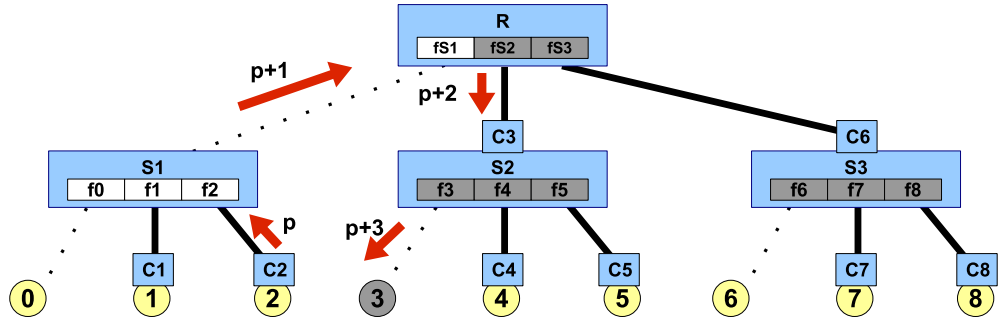
(a) All cores request the lock at the same time (cycle 1).



(b) Lock is granted to Core0 (cycle 4).



(c) Core0 releases the lock (*m*) and *S1* designates Core1 as the next lock holder (*m+1*). Moreover, Core1 releases the lock (*n*) and Core2 becomes the next lock holder (*n+1*).



(d) Core2 releases the lock (*p*) and *S2* designates Core3 as the next lock holder (*p+3*).

Figure 4.4: Example of lock synchronization under the *GLock* mechanism.

round-robin scheduling policy already discussed and would send the TOKEN signal at cycle 3. At cycle 4 and based on the round-robin policy, $S1$ chooses Core0 and sends the TOKEN signal granting access to the lock.

Figure 4.4c shows the scenario in which an Sx can grant the lock ownership without involving any additional notifications to R. More specifically, once Core0 releases the lock at cycle m, its controller sends the REL signal (by writing to the local $f0$ flag, as we mentioned) to $S1$. Next, at cycle $m+1$, $S1$ grants the lock ownership (by means of the TOKEN signal) to the next core by following the round-robin policy from the active fx flags. In this case, Core1 becomes the new lock holder. In the same way, Core2 would be granted the lock in cycle $n+1$ ($m < n$). Finally, in Figure 4.4d we illustrate the scenario when an Si finishes its scheduling because either it has reached the last active fx or there are no more pending local requests for the lock. In this case, Si must send the REL signal towards R, which will choose another available *Sj lock manager* from those that activated the fSx flags. In the figure, $S1$ sends the REL signal to R at cycle $p+1$ ($n < p$), which following the round-robin policy grants the lock to $S2$. Finally, $S2$ sends the TOKEN signal giving access to the lock to Core3 at cycle $p+3$.

4.2.3 Programmability Issues

The *GLock* mechanism proposed in this chapter is intended to be used by programmers in a transparent way. For that, as shown in Figure 4.5, we propose to provide special library-level lock and unlock methods (`GL_Lock` and `GL_Unlock` in the figure) that encapsulate the functionality of *GLock* and that could be used in parallel applications to deal with contended locks. This synchronization method uses a couple of special 1-bit registers added to each processor core. First, the `lock_req` register that is used to request the lock and wait for lock acquisition. Second, the `lock_rel` register that is used to release the lock[2].

As a result of the activation of the `lock_req` register by a processor core, the synchronization protocol explained in the previous section would be invoked. In particular, the corresponding fx flag is activated by the *local controller*, and the secondary and primary *lock managers* start with delivering the lock ownership (granting the token). Straight afterwards, the processor core enters in a loop waiting for the lock ownership (see Figure 4.5). Next, once the lock is granted, the `lock_req` register is reset by the *local controller*, and the core can resume to execute the corresponding critical section protected by the lock. Once the critical

[2]Note that all pairs of flags (one per lock) could be grouped in each core using one special lock register.

```
GL_Lock() {
   asm {
        # Arrival at the CS: set lock_req
        mov 1, lock_req

        # Busy-wait until lock_req is reset
        loop:
                bnz lock_req, loop
   }
}

GL_Unlock() {
   asm {
        # Release lock: set lock_rel
        mov 1, lock_rel
   }
}
```

Figure 4.5: Encapsulating the *GLock* functionality into the lock/unlock library-level methods.

section is executed, the processor core sets the `lock_rel` register that will be used to release the lock. In consequence, the *local controller* would deactivate the fx flag and the `lock_rel` register would be reset as well.

As explained later in Section 4.2.6, the `lock_req` and `lock_rel` registers need as many bits as the number of *GLocks* provided in hardware (one bit per contended lock). In this way, several lock operations involving different sets of cores (the threads in each set running one application) could take place simultaneously. To this end, the register file of each core must be augmented with both registers and the interplay between controllers and them must be enabled, switching on the controllers whenever the `lock_req` registers are written, and switching off the controllers once all `lock_rel` registers are reset and all controllers have unset all the fx and fSx flags.

As pointed out through this chapter, our *GLock* mechanism is aimed at accelerating highly-contended locks. Obviously, the programmer is responsible for identifying locks of this kind and using the `GL_Lock` and `GL_Unlock` methods previously described for them. In the literature, there have been proposed several heuristics to detect contended locks in those cases in which it could be a tedious or difficult task. As an example, Tallent *et al.* [110] have recently proposed

strategies for gaining insight into performance losses due to lock contention. Their goal was to understand where a parallel program needed improving.

As a final observation, the programmability of our *GLock* proposal is orthogonal to the utilization of any optimizations to harness the commented process of busy-waiting to conduct some other useful work while the lock ownership is not granted yet. For example, similar to *try locks* [95], upon a thread requests the lock the thread could execute some alternative code, or as in [47, 78], it could be involved some special queuing and scheduling kernel functions in order to deschedule the waiting thread allowing another one to make progress until the lock is eventually granted. Nevertheless, the implementation of these other approaches does not fall within the scope of this thesis.

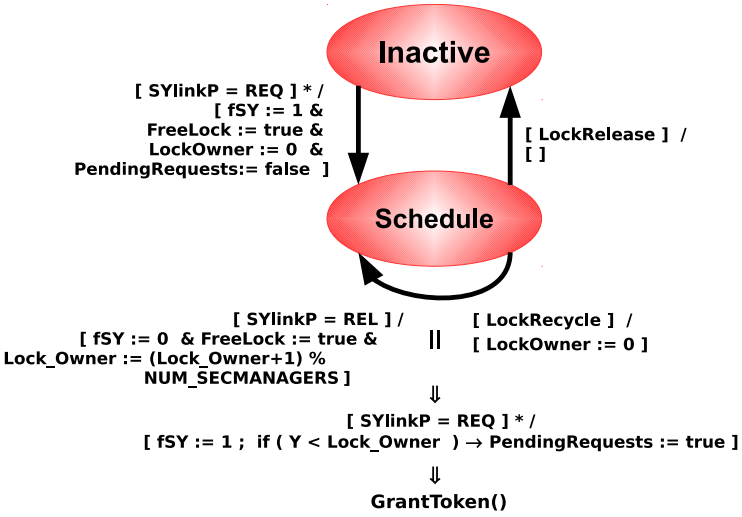
4.2.4 Implementation of *GLock*'s controllers

In this section, we take a closer look at the implementation of the *GLock*'s controllers (see Figure 4.6). As we can see, there are three automata corresponding to each of the three kinds of controllers aforementioned: primary and secondary *lock managers*, and *local controllers*. Over each transition, we depict the event that motivates the transition to the next state and the action that may produce a new event. It follows the pattern: [EVENT] / [ACTION].[3] In more depth, we distinguish the following events and actions:

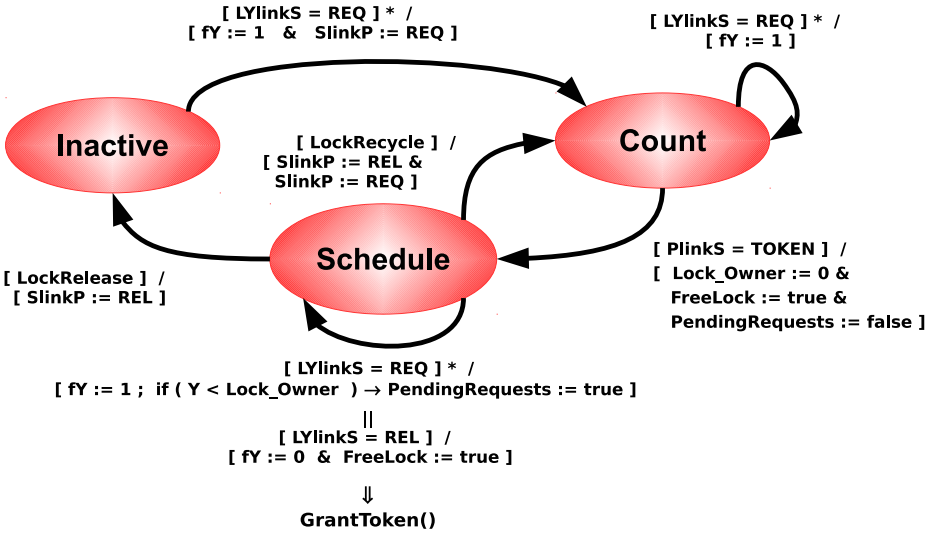
- A core writes the registers `lock_req` or `lock_rel`: e.g. *Core(lock_req:=1)*. Notice that we use := to assign a value, and = to compare two values.

- A controller transmits a signal across a *link*: *TlinkR:=SIGNAL*, where *T* is the transmitter and *R* refers to the receiver controller. Then, *T* and *R* could take the following values: *L*, *S* and *P* for *local controller*, secondary and primary *lock managers*, respectively. Note that, if there is more than one possible candidate to send a particular signal (e.g. in Figure 4.6a, the primary *lock manager* could receive signals from a sort of different secondary *lock managers*), *T* is defined as two letters, the first one refers to the type of controller involved in the transmission (e.g. *S*), whilst the second one, the *Y* letter, represents any of them (e.g. see *SY* in the figure). Besides, SIGNAL identifies the type of signal (REQ, TOKEN and REL) transmitted across the *link*.

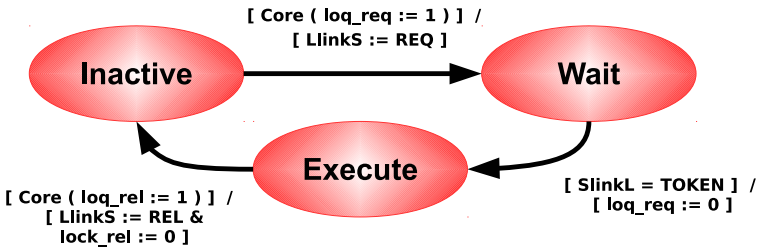
[3] When an event is marked with the $*$ symbol, it means that multiple instances of such an event could happen at the same time.



(a) Primary lock manager automaton.



(b) Secondary lock manager automaton.



(c) Local controller automaton.

Figure 4.6: Finite state automata that implement the *GLock*'s controllers.

```
1  void GrantToken () {
2    if ( FreeLock == true ) {
3         i= Lock_Owner;
4         while ( i < Num_Controllers && FreeLock == true ) {
5           if (  Fi == 1  ) {
6                   Lock_Owner= i;
7                   TlinkCi= TOKEN;
8                   FreeLock= false;
9           }
10          i++;
11        }
12        if (FreeLock == true) {
13          if (PendingRequests == false) => LockRelease;
14          else => LockRecycle;
15        }
16   }
17 }
```

Figure 4.7: `C++`-like pseudo-code for the `GrantToken` function in order to assign the token.

- A controller writes into an internal memory element: *FreeLock:=true* or *LockOwner:=0*. As each controller is made up of both combinational and sequential logics to implement its functionality, some memory elements are required to hold values at every clock cycle (i.e. Flip-Flops). Examples are: *FreeLock*, to specify whether the critical section is free or not; *LockOwner*, for specifying the controller that holds the lock ownership; or *fY*, that stores 1 or 0, depending on whether the *Y* controller has requested or released the lock, respectively.

- The function in charge of granting the lock: `GrantToken()`. For the sake of clarity, the explanation of such a function has been moved to Figure 4.7. As we can observe in this figure, in line 5 we use the *F* letter to refer in general to both *f* and *fS* flags in the automata. Moreover, in line 7, where the TOKEN is granted by *T* controller to the new owner (*Ci*), the pair *T* and *Ci* represents the primary and secondary *lock managers* in Figure 4.6a, and the secondary *lock manager* and *local controller* in Figure 4.6b, respectively. Besides, in lines 13 and 14, we represent two events (`LockRelease` and `LockRecycle`) that motivate new transitions in the two automata aforementioned. Finally, the *Num_Controllers* value refers to the number of secondary *lock managers* or *local controllers* for primary or secondary *lock managers*, respectively.

- A controller has to process a number of events following a specific order: the event situated over the ⇓ symbol has to be processed prior to the event

Table 4.1: Cost of *GLock* for a 2D-mesh CMP layout with R rows and C columns for a total of $R \times C = N$ cores.

#Links	$N-1$
Primary *Lock Managers*	1
Secondary *Lock Managers*	R
Local controllers	$N-1$
`lock_req` **registers**	N (1 bit)
`lock_rel` **registers**	N (1 bit)
***fSx* Flags**	R (1 bit)
***fx* Flags**	N (1 bit)
***FreeLock* registers**	$1+R$ (1 bit each)
***PendingRequests* registers**	$1+R$ (1 bit each)
***LockOwner* registers**	$1(\lceil log_2(R) \rceil$ bits)$+R(\lceil log_2(C) \rceil$ bits each)
Lock Acquire (worst case)	4 cycles
Lock Acquire (best case)	2 cycles
Lock Release	1 cycle

under it. It means that when the two events situated over and under such a symbol in the automata arise at the same time, the precedence order has to be carried out. In addition, if it does not matter what the particular sequential order is, we use the $||$ symbol.

- No event or action: *[]*.

Finally, for the sake of clarity, we omit in the automata when communication is performed locally by means of a flag (see Section 4.2.1). Then, we suppose that all controllers communicate with each other by means of signals through *links* to carry out the synchronization protocol.

4.2.5 Implementation Costs for *GLock*

In this section, we discuss the costs that the *GLock* mechanism would entail considering a 2D-mesh physical layout for the CMP (see Table 4.1). First of all there is the number of *links* that must be used to configure the special network. As already commented on, the on-chip network deploys separate sets of *links* per lock. In particular, each lock needs a set of $N-1$ *links* with N being the total number of cores. In addition, lock synchronization is achieved using a set of controllers, which includes one primary *lock manager*, R secondary *lock*

managers and $N-1$ *local controllers*. Each of these controllers would implement the simple synchronization protocol described in Section 4.2.2. The register file in each core must be extended to provide the `lock_req` and `lock_rel` registers. The total number of bits devoted to both registers is equal to 1, which will depend on the total number of *GLocks* implemented in the system (one bit per lock). Besides, primary and secondary controllers would use a set of R 1-bit fSx and N 1-bit fx flags per lock, respectively. We also show the different internal memory elements employed by *lock managers* explained above (*FreeLock*, *LockOwner* and *PendingRequests*, called registers in the table), along with the respective number of bits that they use. Finally, according to our discussion in Section 4.2.2, the theoretical latency to acquire a lock when nobody has the lock ownership is four cycles in the worst-case scenario, while it takes only two cycles for the best case. To release the lock our mechanism takes a single clock cycle.

The implementation costs given above for our *GLock* infrastructure are the same independently of the two types of technology used (*G-Lines* and *Standard*). As we will expose in Section 4.3.1, they represent a marginal on-chip area overhead and a negligible impact on power dissipation for both technologies. Other important benefits derived from the use of *GLock* are that differently from shared-memory-based locks, our proposal neither consumes space in memory and local caches with synchronization information nor involves the cache coherence protocol. In this way, the *GLock* mechanism would avoid the significant amount of traffic that shared-memory-based locks would introduce in the main data network in high-contention situations. This would also translate into important energy savings for the main CMP's interconnection network, as we will show in Section 4.4.4.

4.2.6 Generalization of the *GLock* Mechanism

In this section we explain how the basic proposal discussed until now (hardware support for just one highly-contended lock) can be generalized so that it can be successfully implemented in several scenarios.

4.2.6.1 Several *GLocks*

Up to now, we have described support for just one lock operation among all cores. However, our mechanism could be easily extended to support a higher number of *GLocks*. For that, the resources required by one *GLock* (and detailed in Section 4.2.5) should be replicated as many times as the number of required

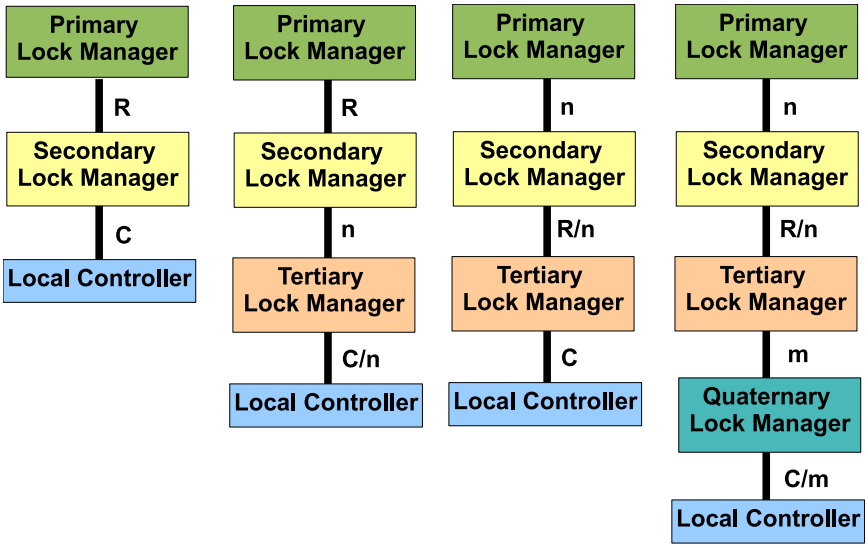


Figure 4.8: Different schemes to incorporate additional levels into the initial three-level hierarchy (the left-most scheme) for an R×C-core CMP with a 2D-mesh network.

GLocks. Since our proposal is specially aimed at highly-contended locks, we think that only a very limited number of *GLocks* would be enough in most cases (e.g. up to two for the real applications evaluated in this chapter). It is worth noting that our proposal is also compatible with the use of software-based lock implementations. This means that even the same application could make use of both *GLock* and software implementations simultaneously. The latter would be the preferred choice for small subsets of cores because in this case the benefits provided by *GLock* would not be significant.

4.2.6.2 Larger Many-Core CMPs

According to Section 4.2.5, our *GLock* infrastructure requires $N-1$ *links* for a 2D-mesh CMP layout. Nevertheless, this method has limited scalability and could not deal with potentially much larger CMPs, mainly for two reasons. First, as the number of cores increases, the maximum operating speed supported by the *GLock*'s *lock managers* would become increasingly slower to be able to sample the signals transmitted across a higher number of *links*. And second, the longer the *link* lengths, the higher the propagation delay will be, thus requiring a higher number of clock cycles to acquire and release a lock.

To overcome such limitations, we propose to extend the hierarchical *GLock* infrastructure outlined in Figure 4.3 by incorporating additional levels of *lock managers* into the initial three-level hierarchy, as we can observe in Figure 4.8. The key rationale behind incorporating these new levels will be to have smaller groups of controllers that a certain *lock manager* needs to monitor, thus simplifying its internal logic, enabling the opportunity to reach higher operating frequencies and setting up shorter *links*. For example, if the number of columns (C) is critical to performance, the second scheme (from left to right in the figure) suggests to incorporate a number of n tertiary lock managers in order to split into n groups the total of C *local controllers* in the *GLock* architecture (n will depend on the exact magnitude of C). Obviously, the more levels in the hierarchy, the more clock cycles will be required to acquire and release the lock. Nonetheless, this apparent drawback will be offset by the higher operating speeds that could be reached. Moreover, when the token is granted to a *lock manager* that is in charge of *local controllers* (e.g. quaternary lock manager in the right-most scheme shown in Figure 4.8), each *local controller* could acquire and release the lock employing the same number of clock cycles as for the initial three-level hierarchy, thus not requiring to take those further steps in some cases. Finally, it is worth noting that incorporating new levels in the *GLock*'s hierarchy does not affect the interface for programmers discussed in Section 4.2.3.

Alternatively, to also achieve high scalability, our proposal could also be easily implemented assuming the leading-edge nanophotonic technology [123].

4.2.6.3 Simultaneous Multithreaded Processor Cores

The *GLock* architecture has been devised to operate on single-threaded processor cores. This section explains how to extend our proposal when simultaneous multithreaded cores (SMT cores) are considered. In this kind of processors, every core's thread has its own register file (or register renaming table) and shares resources such as the functional units among others. The problem in SMT cores is that several threads all belonging to the same core would compete for the same *GLock*'s resources.

If all executing threads in the processor core belong to the same application, every core's thread would indicate the request to acquire the lock through its private `lock_req` register as usually (by setting the bit associated with the lock). But now, an additional step will be required to choose one of the petitioner core's threads as the winner to exclusively activate the core's *local controller*. To do so, we propose to include both a full bit vector in hardware that will keep track of

every petition from the core's threads to acquire the lock, and a round-robin strategy in order to grant the right of using the *local controller* to just only one of the core's petitioners when the lock is released. After that, the *GLock* execution would be the same as for a single-threaded processor core. On the contrary, if the threads executing on the same core belong to different applications, they are forced to use different *GLocks* for highly-contended locks. Finally, since the number of threads per core is commonly very low, not only due to physical constraints but also to memory footprint constraints, only a few *GLocks* would suffice to deal with this worst case scenario.

4.3 Performance Implications

In this section, we analyze *GLock* to determine its potential impact on performance. For that, we start by describing the two types of technologies employed to implement our *GLock* infrastructure. Next, for both implementations, we show their potential contributions to performance in terms of some important raw statistics such as on-chip area overhead, power dissipation, maximum operating speed and minimum latencies for acquiring and releasing a lock.

4.3.1 Implementation Technologies

To implement *GLock*, we leveraged two different technologies. First, we have made use of the state-of-the-art full-custom *G-Lines* technology explained in Section 1.6. Second, we have employed the standard design methodology, described in Section 2.2.3, to achieve a cost-effective *GLock* implementation at the expense of some negligible degradation in performance, as we will see.

4.3.1.1 *G-Lines* Technology

As discussed in the previous chapter, there were several reasons why we decided to use this technology to develop our synchronization mechanism for highly-contended locks in many-core CMPs. First, the connectivity pattern utilized to deploy the dedicated *GLock*'s network (see Section 4.2.1) is based on long 1-bit single-dimension *links* which perfectly fit into the concept of *G-Lines*. Second, according to the results reported in [142], that show negligible area overhead for a 392-*G-Line* network, the 32-core CMP system evaluated in this chapter (further details in Section 4.4.1) is made up of one-12th of the latter number of *G-Lines*, thereby even lower implications for on-chip area will be obtained. This

marginal area overhead will have also a negligible impact on power dissipation. Finally, the *GLock*'s synchronization protocol explained in Section 4.2.2 could take advantage of the extremely fast transmissions at 2.5 GHz that the use of the *G-Lines* technology would entail. In this way, we can directly adopt the same theoretical synchronization latencies for acquiring and releasing a lock presented in Table 4.1.

4.3.1.2 *Standard* Technology

The *GLock* architecture has also been implemented relying on the mainstream industrial synthesis toolflow with an STMicroelectronics 45 nm standard cell technology library are presented in Section 2.2.3. While this standard design methodology leads to cost-effective implementations in the embedded computing domain, low-latency communications for the *GLock*'s *links* are non-trivial to materialize. First, *links* have to be synthesized as RC-based wires[4] that are fully exposed to the effects of technology scaling. More specifically, the RC propagation delay of every wire will degrade as feature sizes shrink, making *links* increasingly slow. For this reason, this technology is also known as an interconnect-dominated nanoscale technology. And second, the propagation delay also affects the internal *GLock*'s logic thus reducing its maximum operating speed.

As for *GBarrier* in the previous chapter, it is worth noting that our mechanism has been synthesized by ensuring minimum wire lengths by situating lock managers in the central row/column of the 2D-mesh layout depicted in Figure 4.2. In addition, we define non-routable obstructions that are placed to mimic the area of every core (550×550 μm^2) of the simulated system explained in Section 4.4.1. Additionally, fences are defined to limit the area where the cells of each *GLock*'s controller can be placed. Such obstructions and fences also ensure minimum-length routing for the *links* in order to reduce their impact on performance and area overhead as the wire length increases.

Due to the fact that RC-based *links* are very critical to performance degradation, we have implemented each *GLock*'s controller by separating the delay that signals take along the wires, from the effective computation that the controllers require to generate their output signals. Notice that, for small many-core CMPs, the critical path that limits the maximum operating speed in our *GLock* infrastructure is defined by the most complex controller (i.e. the *lock manager* which communicates with a higher number of controllers), but as the wire length increases for larger CMPs, the wires could represent such a critical path. Con-

[4]We use the terms *links* and wires interchangeably.

Table 4.2: Raw statistics using *G-Lines* and *Standard* technologies for a single *GLock* in a 32-core CMP layout.

	Freq. (MHz)	**Latency (cycles)**	**Area (μm^2)**	**Power (mW)**
G-Lines	2,500	Acquire: 4 (worst), 2 (best) Release: 1	*Negligible*	28
Standard	714	Acquire: 9 (worst), 5 (best) Release: 3	6,269	*Negligible*

sequently, separating wire delays from controllers delays becomes essential in order to achieve maximum clock speeds. In this way, by using this technology, we cannot directly assume the theoretical synchronization latencies presented in Table 4.1, and a higher number of cycles will be required to acquire and release the lock.

4.3.2 Raw Performance Statistics

Table 4.2 shows the main raw performance statistics obtained from the use of both technologies to implement *GLock*. In particular, we illustrate the maximum operating speed, the latencies of the lock acquisition and release (assuming that the lock is free) and also the area overhead with an estimation of power dissipation that our proposal entails.

The maximum operating speed achieved by the *G-Lines* technology is 3.5 times higher than for the *Standard* technology. Moreover, the number of clock cycles employed by the former technology to acquire and release a lock is half of those achieved by the latter technology. The reason is that every *GLock*'s controller and *link* involved take a different clock cycle in the synchronization process. Therefore, the superior efficiency of *G-Lines* technology reports roughly an eight times faster *GLock* implementation.

Due to the very lightweight infrastructure deployed to implement *GLock*, negligible overheads in terms of die area are obtained for both technologies. Regarding the *G-Lines* technology, as aforementioned, our *GLock* infrastructure requires one-12th of the number of *G-Lines* reported in [142] thus leading to even lower implications for on-chip area. Moreover, as to the *Standard* technology, an area overhead for *GLock* equal to 6,269 μm^2 is reported that corresponds to a negligible 0.07% of the total area employed for the simulated 32-core CMP layout (remember that we assume that each core is 550$\times$550 μm^2 in size).

The latter marginal on-chip overhead must also lead to a negligible impact on power dissipation. We demonstrate this by estimating the power dissipation for a worst-case scenario in which the maximum number of *GLock*'s transmitters and receivers are operating at once. As an example, we detailed the power estimation considering the *G-Lines*-based implementation for *GLock*. According to the *GLock*'s synchronization protocol already described (see Figure 4.4), this situation arises when all cores request the lock ownership at the same time. In this way, for the simulated environment described later in Section 4.4.1, where we considered a 4×8-core CMP[5] , there will be a total of seven local controllers per row (i.e. 28 transmitters) transmitting the 28 REQ signals towards the corresponding four secondary lock managers, which in turn store those signals in the corresponding fX flags (i.e. 28 receivers are required). For the power estimation, we assume the same power dissipation parameters for a 65-nm CMOS process simulated in [142]: 0.6 mW per transmitter; and 0.4 mW per receiver. Moreover, according to [142] no static power is dissipated by the *G-Lines*. Hence, for the number of transmitters and receivers discussed before, the total power estimated is 28 mW (28×0.6 + 28×0.4). It is worth noting that, utilizing CACTI [52], the magnitude of this dissipation is less than one-10th of the power dissipated per read port in the L1 caches simulated in this chapter (see Table 4.3).

As a conclusion of this section, the above results suggest that the fastest technology is the most appropriate implementation to materialize *GLock*. Although synchronization delay would become the discriminating factor, we have also to take into account that the *G-Lines* technology is not within reach of a standard cell design methodology. In consequence, it would be of paramount importance to determine the exact magnitude of such performance degradation when using the *Standard* technology. In case of being negligible, the slower technology would be the preferred *GLock* implementation. This experiment will be conducted in Section 4.4.4.1, by comparing synchronization timings of the two *GLock* implementations in comparison to the best software-based implementation for highly-contended locks.

[5]For simplicity, we assume that 8 cores per row can be materialized in *G-Lines*. Recall that this technology is limited to 7 cores per row and, for example, a 6×6-core CMP layout must be considered instead to span the simulated 2D-mesh 32-core system.

Table 4.3: CMP baseline configuration.

Number of cores	32
Core	3GHz, in-order 2-way model
Cache line size	64 Bytes
L1 I/D-Cache	32KB, 4-way, 2 cycles
L2 Cache (per core)	256KB, 4-way, 12+4 cycles
Memory access time	400 cycles
Network configuration	2D-mesh
Network bandwidth	75 GB/s
Link width	75 bytes

4.4 Evaluation

In this section we give details of our experimental methodology and performance results. For that, the raw performance statistics already discussed in Section 4.3 have been integrated into the simulation environment described in Section 4.4.1. In the latter section, we also describe the sort of benchmarks and their main characteristics utilized to evaluate *GLock*, and a post-morten analysis is carried out in Section 4.4.2 to precisely quantify the exact degree of contention of locks in every benchmark. Moreover, Section 4.4.3 describes the most efficient software implementation for highly-contended locks that *GLock* is compared against. Finally, Section 4.4.4 shows performance results in terms of execution time, network traffic and energy consumption.

4.4.1 Experimental Setup

As *GLock* has been specifically tailored to work in the context of many-core CMPs, we have integrated our proposal into the Sim-PowerCMP performance simulator described in Section 2.2.1. In particular, Table 4.3 shows the values of the main configurable parameters assumed in this chapter. In short, we have simulated a 32-core CMP architecture with an aggressive 2D-mesh network built in a 45 nm process technology.

To evaluate the performance benefits derived from *GLock*, five microbenchmarks and three real applications are used. On the one hand, the microbenchmarks that we have employed are: SCTR, MCTR, DBLL, PRCO and ACTR. Section 2.3 describes each of them. They were chosen because of exhibiting

different highly-contended access patterns to shared data that can be commonly found in parallel applications. On the other hand, regarding real applications, we have considered Qsort sorting algorithm as well as two programs belonging to the SPLASH-2 benchmark suite [128]: Ocean and Raytrace. These applications were chosen since they present a significant lock synchronization overhead due to the existence of highly-contended locks[6]. In fact, these locks are accessed following similar patterns to those of the microbenchmarks. We summarize the characteristics of the microbenchmarks and applications used in this chapter in Table 4.4. For each of them we account for the input size, the total number of different locks, the number of these locks that are highly-contended (H-C Locks), and point out the highly-contended lock access patterns in terms of the microbenchmarks they are similar to.

It is important to note that only contended locks are implemented using the *GLock* mechanism. For the rest of the locks, we rely on a straightforward implementation called *Simple Lock*, that atomically toggles a boolean flag to acquire and release the lock (further details in Section 4.5), that is enhanced with the `test-and-test&set` optimization. This includes the locks used in the applications' library of our simulator to implement barriers. Apart from not being application-level, these locks do not exhibit high contention levels since our simulator provides applications with an efficient tree barrier implementation (up to two threads requesting every lock). In this way, barriers are not affected by our proposal. Finally, all experimental results reported in this chapter are for the parallel phase of all of the benchmarks previously described.

4.4.2 Post-morten Analysis of Benchmarks

To determine the contention of locks, we performed a post-mortem analysis of the benchmarks under study where locks use the *Simple Lock* algorithm enhanced with the `test-and-test&set` optimization. Every time a core tries to acquire a lock, we register the number of concurrent requesters (group of acquiring cores or grAC ranging from 1 to 32) on a cycle-by-cycle basis until the lock is granted to the core. In this way, we can precisely compute each lock's contention rate as the number of cycles where the number of concurrent requesters is equal to each grAC divided by the total amount of cycles where the number of concurrent requesters belongs to the range [1,32]. That is, the lock's contention rate (*LCR*) of a particular lock (*Lock*) for each grAC ($i \in [1,32]$) would be defined by Equation 4.1.

[6]In this thesis, highly-contended locks are those locks accessed by all threads simultaneously or very close in time.

Table 4.4: Configuration of the benchmarks and lock-related characteristics.

Benchmark	Input Size	Locks	H-C Locks	Access Pattern
SCTR	1,000 iterations	1	1	-
MCTR	1,000 iterations	1	1	-
DBLL	1,000 iterations	1	1	-
PRCO	1,000 iterations	1	1	-
ACTR	1,000 iterations	2	2	-
RAYTR	teapot	34	2	SCTR
OCEAN	258x258 ocean	3	1	SCTR
QSORT	16,384 elements	1	1	PRCO

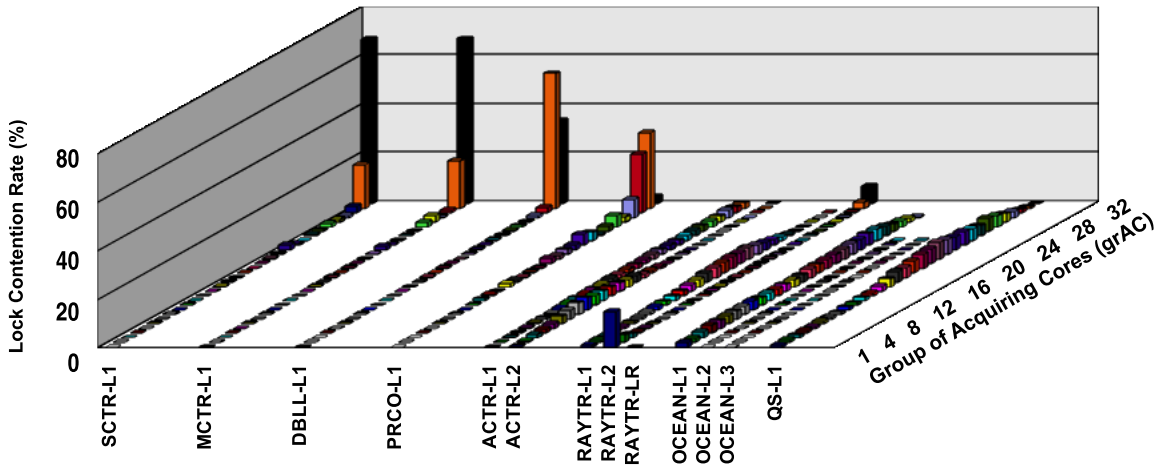


Figure 4.9: Lock contention rate.

$$LCR_{grAC_i} = \frac{Cycles(Lock, grAC_i)}{\sum_{g=1}^{32} Cycles(Lock, grAC_g)} \tag{4.1}$$

In Figure 4.9, the lock's contention rate for all of the benchmarks (x-axis) is shown. In particular, we show the lock's contention rate (y-axis) for all of the possible values of grAC (z-axis). Moreover, we decompose the results for each benchmark on a per-lock basis[7]. To do that, we assume that Equation 4.2

[7]Although Raytrace has 34 locks, we only include the results for the two most highly-contended locks (RAYTR-L1 and RAYTR-L2) and aggregate the rest (RAYTR-LR).

is satisfied and redefine Equation 4.1 as Equation 4.3. That is, every lock's contention rate has also been estimated depending on the amount of clock cycles it uses. From this, we can easily identify in Figure 4.9 those locks that present high contention, and those that although exhibiting high contention are executed during a negligible amount of clock cycles. Due to their very low impact on execution time, the latter kind of locks would be implemented by using the *Simple Lock* algorithm enhanced with the `test-and-test&set` optimization.

$$LCR_{benchmark} = \sum_{i=1}^{Locks} \sum_{j=1}^{32} L_i CR_{grAC_j} = 1 \tag{4.2}$$

$$L_i CR_{grAC_j} = \frac{Cycles(Lock_i, grAC_j)}{\sum_{l=1}^{Locks} \sum_{g=1}^{32} Cycles(Lock_l, grAC_g)} \tag{4.3}$$

As expected, the microbenchmarks exhibit a very high lock's contention rate when grAC is close to the total number of cores. The exception is the ACTR microbenchmark which presents a moderate homogeneous level of contention across all the grAC range. This is mainly due to the barrier synchronization interleaved between the two lock acquisition operations. The real applications also report a behavior similar to that of the ACTR microbenchmark. In this case the reason is their much coarser granularity which spreads the acquire operations throughout the parallel phase. Finally, it is worth noting that Ocean and Raytrace just have one and two highly-contended locks, respectively.

4.4.3 Lock Implementations

To fairly quantify the benefits of our *GLock* mechanism, we consider the case that highly-contended locks found in the benchmarks previously described are implemented by using *MCS Locks*. As we will explain in Section 4.5, *MCS Locks* are one of the most efficient software algorithms for lock synchronization. In particular, *MCS Locks* gracefully manage high-contention situations by having a distributed queue of waiting lock requesters. On the other hand, for the rest of locks (non-contended ones), we employ the *Simple Lock* algorithm enhanced with the `test-and-test&set` optimization due to it has been shown to lead to lower latencies when threads try to acquire a lock without competition. Finally, since the number of highly-contended locks is commonly very small in real applications (up to two in the applications evaluated in this chapter), we assume that two *GLocks* are provided at hardware level.

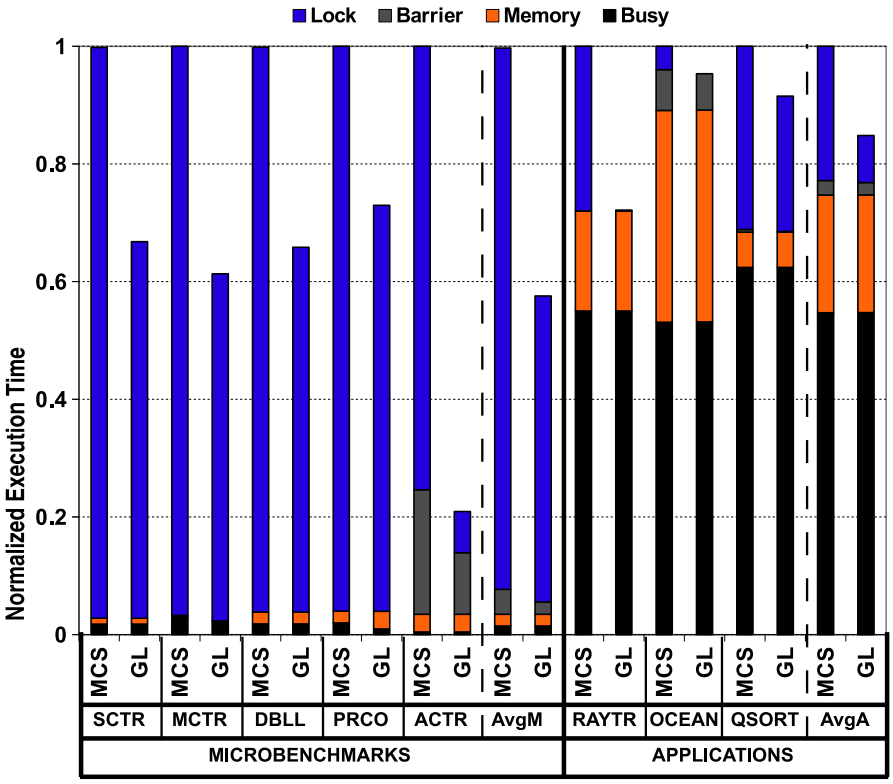


Figure 4.10: Normalized execution time.

4.4.4 Performance Results

The evaluation of the two *GLock* implementations presented in Section 4.3.1 has been carried out taking into account the execution times achieved for the benchmarks shown in Table 4.4, as well as the amount of traffic in the interconnect and the energy-delay^2 product (ED^{2}P) metric for the full CMP.

4.4.4.1 Execution Time

First of all, we consider the implementation of the *GLock* that relies on the *G-Lines* technology. Figure 4.10 shows the execution times that are obtained for the set of benchmarks under study when either *GLock* or *MCS Locks* are employed for the highly-contended locks (GL bars and MCS bars respectively). In particular, execution times have been normalized with respect to those obtained when *MCS Locks* are used. Additionally, each bar shows the fraction of the execution time due to lock and barrier synchronizations (*Lock* and *Barrier* categories respectively),

memory accesses (*Memory* category) and computation (*Busy* category). Finally, average execution times are shown in separate bars for the microbenchmarks (*AvgM*) and applications (*AvgA*).

Regarding the microbenchmarks, we can observe that our proposal presents an average reduction of 42% in execution time (see *AvgM*). The exact extent of the reduction in each case depends on both: the number of highly-contended locks that each microbenchmark has (see Table 4.4), and also the contention rates exhibited by each lock (see Figure 4.9). In particular, our proposal is applied in SCTR, MCTR, DBLL and PRCO to their single contented lock, resulting in reductions of 33%, 39%, 34%, 25% in execution time, respectively. On the other hand, two contended locks are found in the ACTR microbenchmark, which increases the benefits of our proposal (reductions of 81% are obtained). This high reduction is also explained since ACTR presents a much lower contention rate. In particular, in Figure 4.9 we can observe that SCTR, MCTR, DBLL and PRCO present a contention rate close to 80% when considering grACs higher than 20 cores. In contrast, ACTR presents an aggregate contention of only 20% for the same grACs. As we mentioned, *MCS Locks* become inefficient for the low contention case, which accentuates even more the differences between *MCS Locks* and our proposal.

A more in depth analysis reveals that the former reductions come from two kind of effects that the *GLock* mechanism has. First, the time taken to acquire and release the lock is drastically reduced as derived from the improvements shown in the *Lock* category. And second, the fact that our proposal removes from the main data network all extra coherence traffic that a shared-memory-based lock implementation would introduce, also has an effect on the *Barrier* category for the ACTR microbenchmark.

On other hand, the fraction of the execution time that lock synchronization consumes is lower when real applications are considered. In these cases, most of the time is spent on computations and memory accesses (*Busy* and *Memory* categories). This explains the lower reductions in execution time observed for Raytrace, Ocean and Qsort (14% on average). Moreover, since Qsort presents higher contention rates than Raytrace (aggregate contentions of 60% and 29%, respectively, for grACs higher than 20 cores), the *MCS Locks* become more efficient which translates into lower performance differences between *MCS Locks* and the *GLock* mechanism.

Table 4.5 shows speedup results for the real applications (Raytrace, Ocean and Qsort) when scaling the number of cores parameter with the values 4, 8, 16 and 32. Moreover, we use two different lock implementations for the high

Table 4.5: Speedups for the real applications.

Benchmark	Lock Version	4	8	16	32
RAYTR	MCS	3.91	7.53	13.61	20.69
	GL	3.93	7.97	15.67	28.78
OCEAN	MCS	3.70	7.12	13.48	23.62
	GL	3.80	7.32	13.93	25.66
QSORT	MCS	3.67	6.49	9.68	11.38
	GL	3.69	6.55	9.92	12.40

contention case: *MCS Locks* (MCS) and our *GLock* mechanism (GL). From the results shown in Table 4.5, we can extract two important observations. First, all of the benchmarks scale as the number of cores is increased. Second, the exact extent of the speedups depends on the efficiency of the lock implementation we are using. In this way, higher speedups are obtained when employing our *GLock* mechanism which are even very close to ideal speedups in the case of Raytrace.

According to the discussion given at the end of Section 4.3.1.2, it would be of paramount importance determining whether the performance losses in terms of synchronization latency derived from the use of the *Standard* technology can be considered negligible. To this end, Table 4.6 shows the normalized execution times with respect to those obtained when *MCS Locks* are used, depending on the two kind of *GLock* implementations studied in this chapter: *G-Lines*[8] and *Standard* technologies. As we can see, very small performance degradations of 1.6% and 1.3% on average are shown for the microbenchmarks and real applications, respectively. In consequence, we can affirm that our *GLock* mechanism is not so dependent on a full-custom technology to provide extremely efficient synchronizations for highly-contended locks.

Finally, we also carried out a sensitivity analysis to evaluate the extent to which our proposal is affected by longer *link* latencies. To do so, we simulate several configurations of the *G-Line*-based network with varying latencies for the *links* and evaluate the impact that this has on performance. Several clock cycles may be necessary to transmit a signal across one dimension of the chip if, for example, we consider longer *links* that cannot support a propagation delay of a single clock cycle, or even if lower clock frequencies are required to integrate our *GLock* infrastructure in the many-core CMP. Figure 4.11 illustrates

[8]Note that, the results for the implementation that uses *G-Lines* are the same as those presented in Figure 3.10.

Table 4.6: Normalized execution times for *G-Lines* and *Standard* technologies.

	SCTR	MCTR	DBLL	PRCO	ACTR	RAYTR	OCEAN	QSORT
G-Lines	0.67	0.61	0.66	0.73	0.19	0.72	0.95	0.92
Standard	0.68	0.63	0.68	0.75	0.20	0.74	0.96	0.93

the normalized execution times when *G-Lines* take from 1 (results presented in Figure 4.10) to 10 clock cycles (see z-axis in the figure). As we can observe, negligible performance losses are derived even when dealing with 10-cycle *G-Lines*. Particularly, performance degradations of just 1.8% and 1.6% on average in the worst case are shown for the microbenchmarks and real applications, respectively. Note that the results observed for the 10-cycle case are very similar to those obtained for the *Standard* technology previously reported. According to Section 4.3.1, since *Standard*-based implementation is roughly eight times slower than a *G-Lines*-based infrastructure, the former would be equivalent to an 8-cycle *G-Line*-based implementation of the *GLock* mechanism, which explains such similarities.

4.4.4.2 Network Traffic

Our proposal does not generate any coherence messages on the main data network when performing lock synchronizations for any of the two *GLock* implementations. At the end, this translates into the same significant reductions in terms of network traffic. Figure 4.12 shows the total network traffic across the main data network. In particular, each bar plots the number of bytes transmitted through the interconnection network (the total number of bytes transmitted by all the switches of the interconnect) normalized with respect to the *MCS* case. Each bar is broken down into three categories: *Coherence* corresponds to the messages generated by the cache coherence protocol (e.g. invalidations and Cache-to-Cache transfers); *Request* comprehends messages generated when load and store instructions miss in cache and must access a remote directory; and finally, *Reply* involves the messages with data.

For the microbenchmarks, important reductions in network traffic are achieved (76% on average). In general, these reductions are directly related to the extents of the improvements in execution time previously reported. Moreover, since the simulated L2 cache is shared among the different processing cores, but it

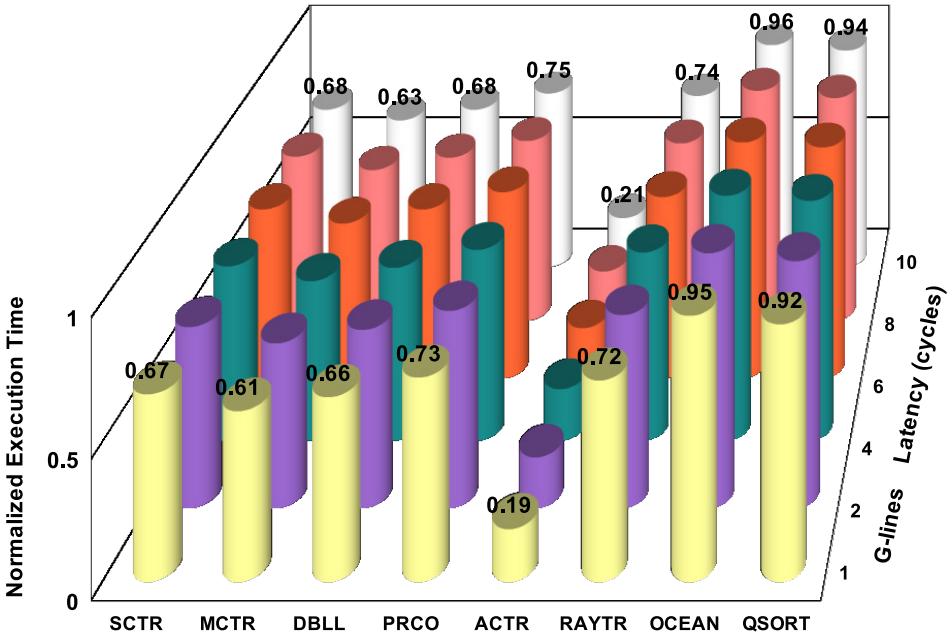


Figure 4.11: Normalized execution times of benchmarks depending on *G-Lines* latency running on a 32-core CMP.

is physically distributed between them (see Section 4.4.1), some accesses to the L2 cache will be sent to the local slice while the rest will be serviced by remote slices. This will also affect to lock acquisition and release operations timings. In contrast, since our *GLock* proposal skips the memory hierarchy we have not obtained such negative impact on network traffic. In particular, SCTR, MCTR, DBLL and PRCO show reductions of 81%, 99%, 72% and 46%, respectively. This is due to the fact that almost all network traffic of these microbenchmarks is due to lock synchronizations. The exception is ACTR, where the barrier used in between the two phases also generates network traffic. However, since the barrier time is approximately 20% of the lock time (see *Barrier* and *Lock* categories in Figure 4.10), a reduction of 80% in network traffic is obtained.

Finally, regarding the real applications, we can see an average reduction of 23% in network traffic (see *AvgA* in Figure 4.12). More specifically, the applications Raytrace, Ocean and Qsort present reductions of 23%, 1% and 45%, respectively. As before, there is a correlation between the fraction of the execution time devoted to lock synchronization and the amount of network traffic that is saved. For

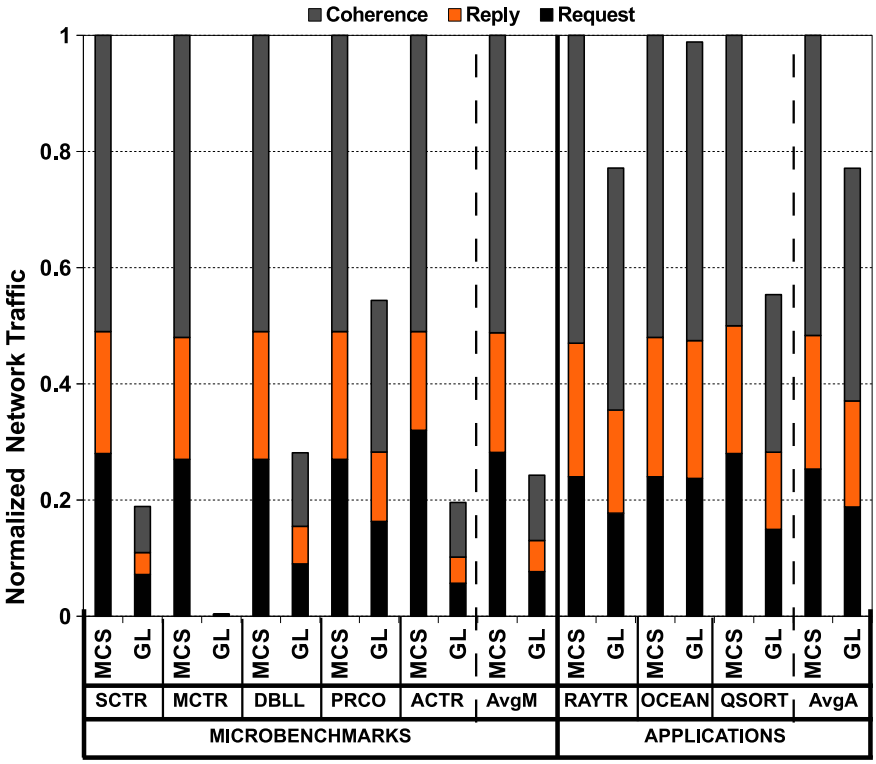


Figure 4.12: Normalized network traffic.

instance, Ocean presents the lowest reduction in network traffic since less than 5% of its execution time (see Figure 4.10) is spent on locks.

4.4.4.3 Energy Efficiency

Finally, we also consider the benefits in energy efficiency that our proposal entails. More specifically, we present in Figure 4.13 the normalized energy-delay^2 product (ED^{2}P) metric for the full CMP. To account for the energy consumed by the *GLock* architecture (the *G-Lines*-based network described in Section 4.2.1), we extend the Sim-PowerCMP with the consumption model of *G-Lines* and controllers described in previous Section 4.3.2. According to our discussion in Section 4.3.1.2, the power dissipation associated with our two technology-aware *GLock* implementations is negligible, hence the power statistics presented in this section will be mainly due to the improvements in execution time and network traffic reported in previous sections.

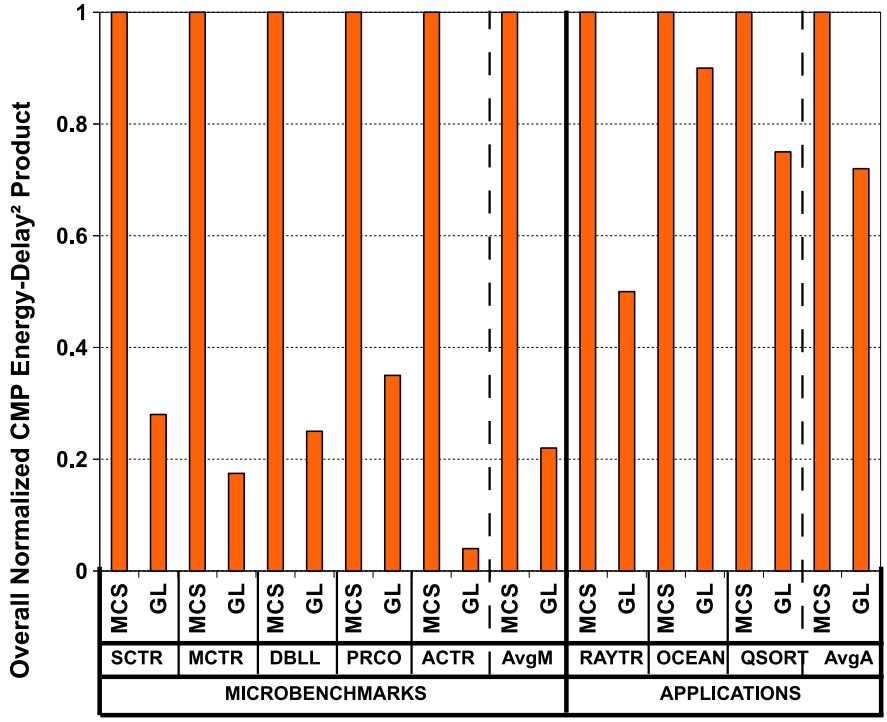


Figure 4.13: Normalized energy-delay^2 product (ED^{2}P) metric for the full CMP.

As in the previous two sections, all results in Figure 4.13 have been normalized with respect to the *MCS* case. As can be observed, important improvements in the ED^{2}P metric of the whole CMP are achieved when applying our proposal. In particular, the *GLock* mechanism brings average improvements in ED^{2}P of 78% and 28% for the microbenchmarks and real applications, respectively. The SCTR, MCTR, DBLL, PRCO and ACTR microbenchmarks show reductions of 72%, 83%, 75%, 65% and 96%, respectively. Additionally, reductions of 50%, 10% and 25% are achieved for Raytrace, Ocean and Qsort.

In general, as commented above, the magnitude of these savings is directly related to the extents of the improvements in execution time and network traffic previously reported. We have found that when the *GLock* mechanism is employed, the number of instructions executed per lock acquisition and release operation is drastically reduced. Note that while *MCS Locks* must deal with a distributed queue of waiting threads requesting the lock, *GLock* only needs two assignment instructions on two registers to notify the arrival to the lock and the subsequent release operation (see Section 4.2.3). Obviously, less instructions executed means less energy consumed in the processor cores.

Moreover, since we reduce the latency of lock acquisitions, the busy-wait process is also shortened with *GLock*. While busy-waiting, a processor core repeatedly access the L1 cache to check the value of a shared variable. In this way, shorter busy-waiting implies less accesses to the L1 cache, and therefore, less energy consumed in this structure. Finally, given the fact that our proposal skips the memory hierarchy, we save all the energy derived from coherence activity when locks are executed. In particular, we remove all of the L1 cache misses related to lock operations and the corresponding messages transferred across the interconnect. This brings reductions in the energy consumed at the L2 cache banks and the interconnection network.

4.5 Related Work

Performance degradation of software-based schemes for lock/unlock operations in parallel machines has long been recognized as a key impediment to scalability and high performance as the processor/core count increases. For that reason, in the literature there have long been devised some architectural extensions that go from simple hardware support, such as improved network/memory controllers, to those proposals that integrate sophisticated interconnection networks for conveying synchronization traffic.

A comprehensive description of the major proposals for lock/unlock operations at both software and hardware levels are described below. To this end, we firstly give a review of some well-known software-based implementations exposing their main performance bottlenecks in order to understand why hardware support becomes essential. Secondly, we expose the most relevant hardware-based schemes by comparing them against our *GLock* proposal.

The simplest software-based synchronization algorithms rely on atomic read-modify-write instructions, such as `test&set`, `fetch&operation`, `swap` or `compare&swap`, to implement the lock and unlock synchronization primitives [29]. For instance, *Simple Lock* repeatedly tries to acquire the lock by toggling a boolean flag from false to true with a `test&set` instruction. Next, the lock is released by simply toggling the flag back from true to false. The main drawback of this algorithm is the continuous generation of cache-coherence network traffic while busy waiting for lock acquisition. To ameliorate this problem two optimizations, namely `test-and-test&set` and exponential back-off, have been proposed. The former issues standard loads that hit on the local cache while busy waiting for lock acquisition. Hence, the `test&set` is only issued when the lock appears to

be free thus reducing cache-coherence network traffic. The latter inserts a delay between consecutive attempts to acquire the lock in order to reduce contention. Anderson [141] found that exponential back-off is the most effective form of delay. Nevertheless, as contention increases these improvements are not enough to guarantee scalability especially for highly-contended locks.

More elaborated algorithms such as *Ticket Lock*, *Array-based Lock* and *MCS Lock* provide more scalable and fair lock implementations at the expense of increased storage cost and higher latency for the low contention case [29]. The *Ticket Lock* algorithm consists of a pair of counters, a *ticket* counter and a *now-serving* counter. To acquire a lock a thread gets its turn by issuing a `fetch&increment` on the ticket counter and then busy waits until the now-serving counter equals its ticket. To release the lock a thread simply increases the now-serving counter. *Array-based Lock* just replaces the now-serving counter by an array of locations. The idea behind *MCS Locks* [76] is similar to that of *Array-based Locks*. An *MCS Lock* builds a distributed queue of waiting threads requesting the lock. In this way, each thread busy waits on a unique, locally accessible flag rather than competing for a single counter. *MCS Locks* are considered the most efficient software algorithm for lock synchronization [6,59,76]. In all three cases, cache-coherence network traffic is reduced because only one thread actually attempts to obtain the lock when it is released by the previous owner.

In general, simple algorithms tend to be fast under low contention and inefficient when contention is high. In contrast, sophisticated algorithms specifically designed to deal with contention usually incur a non-negligible overhead when there is little contention. For this reason, a number of hybrid approaches have been proposed. *Reactive Lock* [14] is a library-based adaptive approach that chooses the best synchronization algorithm under different levels of contention. This technique switches between *Simple Lock* and *MCS Lock* for the low and high contention cases, respectively. *Smart Lock* [59] uses heuristics and machine learning to choose the most appropriate algorithm following a specific user-defined goal in terms of performance, energy consumption or problem-specific criteria.

A completely different software approach that is not based on atomic read-modify-write instructions called *MP-Locks* is presented in [19]. With *MP-Locks* synchronization operations are implemented using message passing, over the main data network, and embedded kernel lock managers. This approach comes in three different flavors, namely centralized, distributed and reactive that are differentiated from each other in how the lock managers control lock ownership. A comparison between *MP-Locks* and *MCS Locks* reports significant performance and scalability gains at the expense of increased software complexity and limited

portability. A similar idea, proposed in the context of distributed systems, called *Token-based Locks* appears in [25]. In this case, the right to acquire a lock is represented by a token which is unique in the whole system. Threads willing to acquire a lock must wait for token arrival and release the token upon critical section completion.

Remote Core Locking [80] (RCL) is an efficient software-based implementation specially designed for highly-contended locks. RCL replaces lock acquisitions by remote procedure calls (RPCs) to a dedicated server core in order to exploit cache locality. The reason is that when a CS accesses shared data that has recently been accessed by another core, there will result in cache misses. So, the idea is to avoid these cache misses. RCL entails a tool that transforms CS code to be executed as an RPC as well as a runtime for Linux OS that includes the RCL code. The implementation of RCL is based on an array of requesting cores (clients) cached in the server core, and devoted to establish an interaction with the server that quits when the server executes the CS. For a 48-core machine significant performance improvements are shown. Nevertheless, the efficiency of this software implementation for highly-contended locks may be hampered by higher core counts, since dedicating an entire core to implement a CS is a centralized approach that may lead to potential performance bottlenecks as the number of clients increases. Differently, our *GLock* proposal does not dedicated cores to execute a CS and is based on a scalable and distributed infrastructure to implement highly-contended locks. In addition, our proposal neither injects synchronization-related traffic into the main interconnect nor uses the memory system, thereby not interfering with QoS of parallel applications.

Hardware support for lock synchronization has also been the target of a number of proposals. Queue-On-Lock-Bit (QOLB) [6] is based on a distributed queue of waiting threads requesting the lock. Unlike *MCS Locks*, in QOLB the queue is implemented entirely in hardware at the cache controller level. Moreover Carter et al. [57] propose to integrate some basic atomic instructions at the directory controller level. The Synchronization-operation Buffer (SB) [100] is a hardware module which augments the memory controller to queue and manage lock operations issued by the threads. QOLB reports non-negligible performance gains when compared to *MCS Locks*. In general, all of the hardware-supported solutions require modifications at some level of the memory hierarchy. In contrast, our proposal, namely *GLock*, completely decouples lock synchronization from any kind of memory-related activity, by deploying a dedicated lightweight on-chip network infrastructure to implement a simple synchronization protocol aimed at accelerating highly-contended locks.

4.6 Conclusions

Lock contention is recognized as a key constraint to performance and scalability on many-core CMPs when trying to exploit thread-level parallelism. In this chapter we have proposed *GLock*, a new hardware-supported implementation for highly-contended locks. *GLock* deploys a dedicated on-chip network and relies on a simple token-based messaging-protocol in order to provide an extremely efficient and completely fair lock implementation. Even though resources required to build this network grow with the number of supported locks, a deep analysis of some parallel applications discloses a reduced number of highly-contended locks in most cases. In this sense, our proposal is a hybrid approach which combines the devised *GLock* mechanism with *Simple Locks* enhanced with the `test-and-test&set` optimization. While *GLock* provide lightning-fast lock acquisition and release for highly-contended locks, the *Simple Locks* result in the best performance for low-contended locks.

We have evaluated two implementations of the *GLock* mechanism. The first makes use of state-of-the-art full-custom technology, namely *G-Lines*, whilst the second is based on a mainstream industrial toolflow and standard cells. While the on-chip area overhead and energy consumed by both implementations are considered negligible, the former technology has been used to report minimum synchronization latency, whereas the latter leads to a cost-effective implementation because it is within reach of a standard cell design methodology.

We integrate both *GLock* implementations into a detailed architectural-level power-performance simulator, and discuss synchronization efficiency results as compared to the most efficient software-based lock implementation. To do so, we simulate a 32-core CMP with a 2D-mesh data network and employ a set of microbenchmarks and real applications. From this study, both *GLock* implementations report very similar reductions in execution time, thus not making our proposal so dependent on a full-custom technology to achieve extremely efficient synchronizations for highly-contended locks. In more depth, an average reduction of 42% and 14% in execution time, an average reduction of 76% and 23% in network traffic, and an average reduction of 78% and 28% in the energy-delay^2 product (ED^{2}P) metric for the full CMP are achieved, for the microbenchmarks and the real applications, respectively.

Chapter 5

ECONO: A Simple and Efficient Cache Coherence Protocol

5.1 Introduction and Motivation

The exploitation of thread-level parallelism in many-core CMPs is commonly carried out by relying on an intuitive shared-memory programming model [31]. This is due to the fact of achieving lower programming complexity which is of paramount importance as core count, and number of concurrent threads to manage, increases. In this memory model, communication and synchronization among threads are accomplished through memory operations over shared memory blocks, such as conventional loads and stores instructions, as well as atomic read-modify-write instructions like *test&set* or *fetch&add*. Nevertheless, the maintenance of coherence for such blocks across all levels of a memory hierarchy, composed of private and shared levels of caches, requires the implementation of a cache coherence protocol.

Strictly speaking, a cache coherence protocol ensures coherence in a shared-memory system as long as two different invariants are maintained [31]. First, the *single-writer&multiple-reader* invariant, where for any memory block A, at any given (logical) time, there exists only a single core that may write to A (and can also read it) or some number of cores that may only read A. And second, by dividing the memory block's lifetime into different epochs, in which either a single core has read-write access or some number of cores (possibly zero) have read-only access, the coherence protocol must also maintain the *data-value*

invariant. The latter guarantees that the value of the memory block at the start of an epoch is the same as the value of the memory block at the end of its last read-write epoch.

Although scalability of coherence is realizable in future many-core CMPs [97], as core count increases achieving efficiency constitutes a great challenge for several reasons. On the one hand, the more the block sharers the costlier the coherence activity will be required. For instance, when invalidating all the copies of a particular cache block that is present in every cache, after a particular thread suffers a write cache miss. On the other hand, apart from high performance, an efficient coherence protocol should also deal with other important aspects, including resulting complexity and requirements in terms of on-chip area and energy consumption.

To exemplify the difficult decision-making process to address the previous aspects at once, we choose two contemporary coherence protocols, namely *Hammer* [2] and *Directory* [88]. In short, *Hammer* ensures coherence by relying on broadcasting coherence messages to all private caches, whereas *Directory* keeps track of coherence information about sharers to send coherence messages just to those private caches with a valid copy of the memory block. Therefore, *Directory* is more efficient in terms of performance and power dissipation since it only injects the required coherence messages into the CMP's interconnection network. Besides, *Hammer* is more efficient in terms of on-chip area resources because it does not devote any hardware structure to store coherence information. In addition, increasing performance levels in these protocols come at the expense of implementing sophisticated state machines with many states. This translates into higher protocol complexity, hence making protocol design and verification much harder [37].

In this chapter, we propose *Express Coherence Notification* (*ECONO*), a cache coherence protocol specifically tailored to future many-core CMPs. The novelty of our proposal resides in that it provides a very efficient operation for coherence maintenance in terms of design complexity, on-chip area overhead, performance and energy consumption. To accomplish this, the design of our proposal is based on *atomic broadcasts* over a *dedicated lightweight on-chip network*, thereby removing a considerable amount of coherence-related traffic from the main CMP's interconnect. As compared to *Hammer* and *Directory*, *ECONO* features the simplest design, requires an on-chip area overhead similar to *Hammer*, reports similar performance to *Directory* and constitutes the most energy efficient design.

The rest of the chapter is organized as follows. We describe *Hammer* and *Directory* protocols in Section 5.2. Section 5.3 presents our proposal for a simple

and efficient coherence protocol for many-core CMPs. Next, in Section 5.4 we discuss some important performance implications when using our proposal. We evaluate the benefits of *ECONO* in Section 5.5. The related work is described in Section 5.6 and finally, Section 5.7 presents the main conclusions of our work.

5.2 Two Contemporary Coherence Protocols

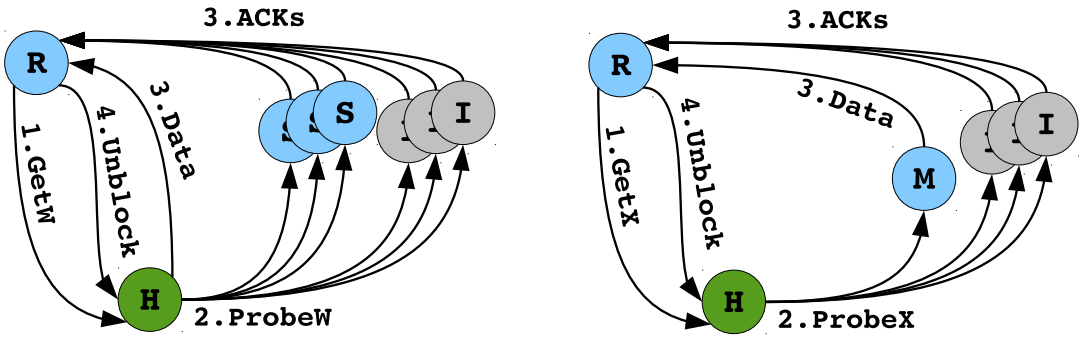
A great deal of attention has been devoted to solve the problem of maintaining cache coherence in an efficient manner for shared-memory CMP systems (see Section 5.6). In this section, we study two contemporary designs for maintaining coherence, *Hammer* and *Directory*, and present how they manage to ensure coherence. For the explanations, we assume a many-core CMP architecture composed of a number of replicated tiles where each tile contains a private L1 cache and a slice of a shared L2 cache (further details in Section 5.3.3). Moreover, as baseline implementation we consider MESI state machines for the L1 caches.

5.2.1 Hammer

This is the cache coherence protocol used by AMD in their Opteron systems [2]. This protocol does not store any coherence information about cached blocks held in the private caches and can be understood as a directory-based protocol without directory information, or using the terminology from Agarwal et al. [1], a Dir_0B protocol. To ensure cache coherence, *Hammer* relies on coherence messages that are broadcast through the main CMP's interconnection network.

In Figure 5.1, we show how *Hammer* would operate under two typical scenarios[1]. The first scenario is depicted in Figure 5.1a. The figure shows the case of a particular memory block that is being shared among different L1 caches (multiple copies of the block, each one in `S` state), and a requesting core (`R`) that gets a write miss in its L1 cache and sends a write request (`1.GetW`) to the home tile (`H`). In this case, the proper coherence action would be the invalidation of all cached copies before delivering the valid home's copy along with write permission to the requester. Since, no coherence information about block sharers is stored in this protocol, this protocol broadcasts as many invalidation coherence messages as the number of L1 caches in the system (`2.ProbeW`). In this way, some L1 caches that receive the coherence message may not contain a valid copy of the requested block (see `I` in the figure). In any case, all the L1 caches respond with

[1]Messages depicted together with the arrows travel through the main CMP's interconnect.



(a) Invalidation of L1$ sharers.

(b) Data recovery from L1$ owner.

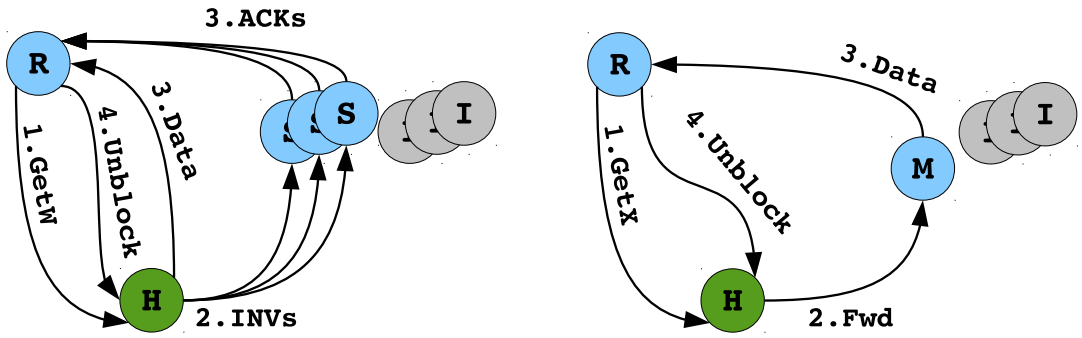
Figure 5.1: Examples of coherence scenarios under the *Hammer* protocol.

acknowledgement messages (`3.ACKs`) to the requesting core. Moreover, after the broadcast transmission has been completed, the home tile can send its block copy to the requester (`3.Data`). Next, once the requester receives all the responses, it sends an unblock message (`4.Unblock`) to the home tile so that the home can attend other requests for the same memory block.

In Figure 5.1b, we show the actions performed in the second scenario. Here, there exists only one modified copy of the block in a single L1 cache (i.e. the owner or `M` in the figure) and a requesting core that wants to write or read (see `1.GetX` for the general case) the block. On the one hand, in case of a write cache miss (i.e. `1.GetW`), once all coherence messages arrive at all L1 caches (`2.ProbeX`), all but the owner respond with acknowledgements (`3.ACKs`) to the requester, whereas the owner tile sends the modified block to the requester (`3.Data`) and invalidates its copy. Then, upon receiving all messages from all other tiles, the requester sends to the home the unblock message (`4.Unblock`). On the other hand, for a read cache miss (i.e. `1.GetR`), the owner would not invalidate its copy of the memory block but a downgrade action would be carried out. As a result, the `M` state of the owner's block would change to `S` and then, the owner would become a new sharer of the block.

5.2.2 Directory

The *Directory* protocol assumed in this chapter is similar to the intra-chip coherence protocol used in Piranha [88]. Differently from *Hammer*, this protocol stores coherence information about the blocks held in the private caches (e.g. through



(a) Invalidation of L1$ sharers.

(b) Data recovery from L1$ owner.

Figure 5.2: Examples of coherence scenarios under the *Directory* protocol.

a full-map sharing code [91]), and manages coherence in a precise manner by not relying on broadcasting for every coherence action performed. Following the terminology from Agarwal et al. [1], this protocol can be classified as a Dir_nNB if every directory entry can store the identity of up to n sharers (being n the number of private caches in the system). For instance, using a full-map sharing code, n would be equal to the number of L1 caches in the system (i.e. a bit-vector with one bit per L1 cache). To illustrate how *Directory* operates to ensure coherence, we use the two scenarios previously described for the *Hammer* protocol.

On the one hand, when there are multiple sharers of a memory block and a requesting core wants to write into it (see Figure 5.2a), the requester gets a write cache miss and a write request is sent to the home tile (`1.GetW`). But now, the coherence information stored in *Directory* prevents from sending unnecessary invalidation messages towards those L1 caches that do not have a valid copy of the block (`2.INVs`), hence saving traffic in the main CMP's interconnect, power dissipation and also latency of cache misses. As we can observe, the rest of the process is the same as shown for the *Hammer* protocol.

On the other hand, when there is only a single valid copy of the block in a private cache (see Figure 5.2b), *Directory* recovers the data for the requesting core following a more efficient strategy. Unlike *Hammer*, rather than sending broadcast messages to all the L1 caches, *Directory* benefits from the coherence information stored for the particular memory block and then, the home tile forwards the coherence message just to the corresponding block owner (`2.Fwd`). In this way, the owner sends the data to the requester (`3.Data`) and, differently from *Hammer*, no acknowledgement message is required from the remaining L1 caches. Once the data arrives at the requester, it sends the unblock message to directory as for

the *Hammer* protocol. Finally, in a read cache miss, the operation would be the same as before but a downgrade action would be performed.

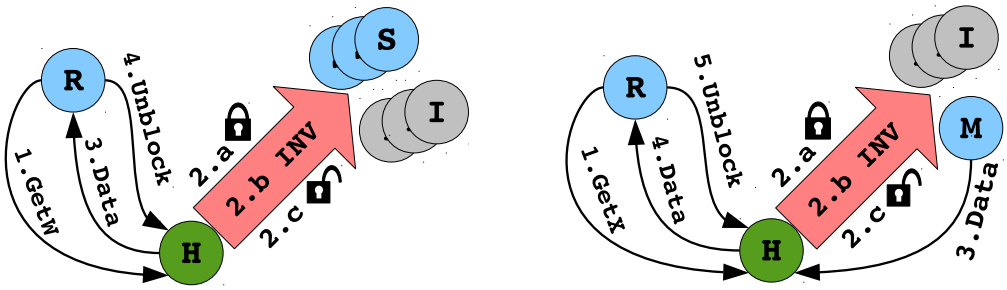
5.3 *ECONO* Coherence Protocol

In this section, we present the *Express Coherence Notification* protocol (*ECONO* from now on), our proposal for a simple and very efficient cache coherence protocol in many-core CMPs. *ECONO* does not keep track of any sharing information and, to ensure coherence, it relies on extremely fast *atomic broadcasts* that are transmitted over a lightweight *dedicated on-chip network*.

The rationale behind our decisions for the design of our proposal comes from the following observations. First, we decided to include atomicity as part of the normal operation of our coherence protocol. In consequence, we could obtain simpler protocol specifications which is of paramount importance to reduce design complexity and protocol verification [37]. Second, to develop a cost-effective coherence protocol, we took advantage from the minimal area overhead required by the *Hammer* protocol, that does not devote any hardware structures to keep track of coherence information about cached blocks. This is the reason why our proposal also makes use of broadcasting to accomplish the coherence actions. Consequently, following the terminology from Agarwal et al. [1], like *Hammer*, our proposal can be also classified as a Dir_0B protocol. And third, we decided to convey all broadcast messages through a dedicated on-chip network in order to avoid compromising QoS of applications. It is worth noting that, neither *Hammer* nor *Directory*, nor the vast majority of the protocol designs operate in this way. Moreover, due to the fact that coherence messages are commonly very short (i.e. the combination of the requested block's address and operation code could be enough), *ECONO*'s network features a very lightweight and low-bandwidth infrastructure to minimize its impact on on-chip area as much as possible.

5.3.1 Baseline Operation

To illustrate how *ECONO* operates, we assume the same CMP architecture used for the descriptions of *Hammer* and *Directory* in previous Section 5.2. Moreover, as a baseline implementation, we make use of a very simple *request-response* operation mode in which home tiles do not delegate coherence responses to L1 caches (i.e. no forward messages, that would increase complexity at the L1 caches, and directory controllers are employed). Additionally, every coherence message



(a) Invalidation of L1$ sharers.

(b) Data recovery from L1$ owner.

Figure 5.3: Examples of coherence scenarios under the baseline *ECONO* protocol.

atomically broadcast (*Atomic Coherence Notification* message, or *ACN* message for short) contains all the information required for the coherence operation: the address of the requested block and the type of the *ACN* message (e.g. invalidation or downgrade). Finally, to illustrate how our proposal would operate, we utilize the same two scenarios previously employed for the explanations of *Hammer* and *Directory*.

For the first scenario, where we deal with a write cache miss and there are multiple sharers of a memory block (see Figure 5.3a), a coherence operation similar to what is done in *Hammer* should be performed due to the fact that *ECONO* does not store any coherence information about cached blocks (no directory exists). However, unlike *Hammer*, our protocol operates much more efficiently in this case for two reasons. First, rather than transmitting in broadcast as many invalidation messages as L1 caches in the system, *ECONO* sends a single *ACN* message over the special on-chip network. And second, no acknowledgment messages are required because we guarantee that the *ACN* message arrives at all L1 caches after a certain amount of time. To accomplish that, the special network was built to enable this type of efficient transfers (further details in Section 5.3.3.2), and every transference is performed atomically with respect to any other coherence operation in the system to impede delays. For that, before starting a coherence action, the particular home tile must acquire the use of *ECONO*'s network in mutual exclusion (see action `2.a` in the Figure 5.3a). Then, the *ACN* message is transmitted in broadcast to invalidate all the L1 caches (action `2.b`). Next, after a certain amount of time (the time required to reach all the L1 caches), the home knows that all cached blocks have been removed and it can release *ECONO*'s network (action `2.c`). Besides, the home sends its

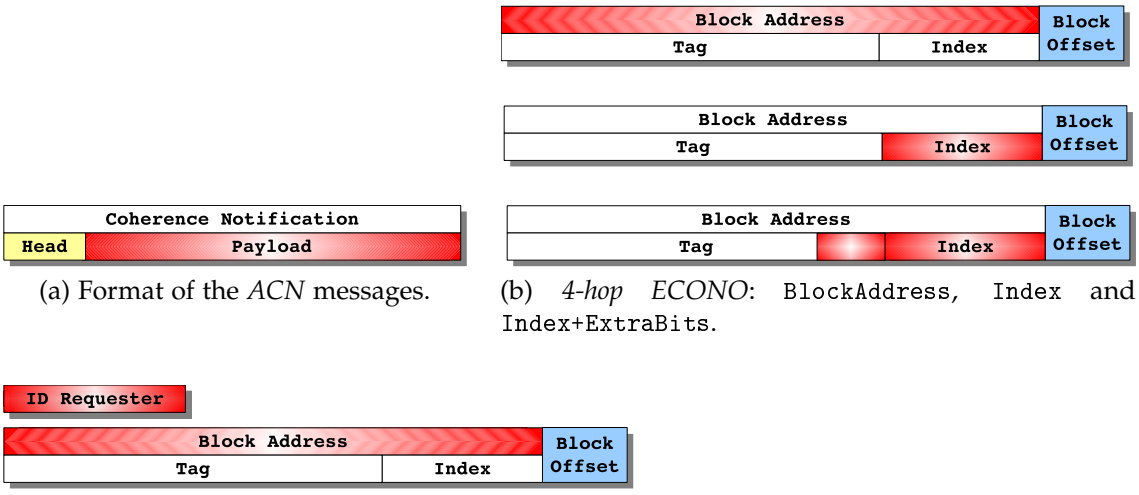
copy of the memory block to the requester (`3.Data`). Upon receiving the data, the requesting core transmits the unblock message to the home tile (`4.Unblock`) as explained for the other two protocols.

In Figure 5.3b, we show the actions performed in the second scenario. Remember that, there exists only one modified copy of the block in a single L1 cache and a requesting core that wants to write or read the block. As we can observe, the process would be the same as before but now, the owner would invalidate (for a write miss) or downgrade (for a read miss) its copy and it would send the block to the home tile (`3.Data`). Next, the home would send the block to the requester (`4.Data`). Finally, the requester would send the unblock message to the home (`5.Unblock`).

An initial analysis of the baseline *ECONO* operation reveals the following conclusions. First, the use of *ACN* messages should report important benefits in terms of network traffic, since neither acknowledgements nor multiple coherence messages are injected into the main interconnection network. Second, it also should provide benefits in execution time because, as explained above, the *ACN* operation is performed very fast and besides, removing all the latter coherence messages from the interconnect should lead to lower periods in blocking states at home tiles. And third, the use of atomic coherence messages helps to simplify the protocol mainly for two reasons. On the one hand, atomicity prevents from considering non-trivial situations where there exists coherence interferences among conflicting requests. On the other hand, it also simplifies protocol verification because serializing coherence actions ensures forward progress that prevents from pathological livelocks or possible starvation scenarios. Consequently, we can affirm that our proposal is simpler than *Hammer* and *Directory*, and also more efficient in terms of network traffic since we remove an important amount of coherence-related traffic from the main interconnection network. However, at this point it is difficult to affirm that we also outperform both protocols in terms of execution time, because we cannot determine the exact impact on execution time of contention when using *ECONO*'s network, and the effect of serialization in the coherence operations. Later on, we will quantify these issues along with others such as power dissipation and on-chip area overhead to complete all the different aspects considered in Section 5.1.

5.3.2 Extensions to the Baseline *ECONO*

The baseline *ECONO* implementation previously presented makes use of the *request-response* operation mode. As explained above, this involves up to four hops



(a) Format of the *ACN* messages.

(b) *4-hop ECONO*: `BlockAddress`, `Index` and `Index+ExtraBits`.

(c) *3-hop ECONO*: `ID+BlockAddress`.

Figure 5.4: Format of the *ACN* messages for both implementations of *ECONO*.

in the critical path of a cache miss (see Figure 5.3b). We call this implementation *4-hop ECONO*. Like *Hammer* and *Directory*, we could reduce the number of hops to three by using forward messages from the home tiles to the owner caches (see Figures 5.1b and 5.2b). In this way, the owner cache would be in charge of sending the data directly to the requester in exchange for a slight higher complexity in the L1 caches. In Section 5.5, we will also evaluate this alternative implementation of *ECONO* protocol, namely *3-hop ECONO*.

Moreover, we could have opted to reduce the blocking periods at the home tiles by removing unblock messages, however we did not choose this option and preferred the use of unblock messages mainly for two reasons. First, the unblock message is required for non-ordered interconnection networks, like the 2D-mesh network assumed in this chapter (further details in Section 5.5.1), in order to handle races in a simple way [32]. And second, the unblock message transmitted towards the home tile ensures that no subsequent *ACN* operation to the same block address is performed in the interim. Therefore, we avoid dealing with possible data races between different *ACN* messages to the same block address. For instance, in Figure 5.3a when the `3.Data` message has not arrived yet and a subsequent *ACN* message is received.

The final group of extensions comes from modifications to the *ACN* messages transmitted. Figure 5.4a illustrates the format of an *ACN* message. As we can see, it is made up of two different fields: the *head*, that is used to identify the

type of action to be applied (e.g. invalidation); and the *payload*, that stores the information required to identify the cached blocks that will be affected by the coherence action (e.g. the block address).

Regarding the baseline *4-hop ECONO*, according to Section 5.3.1, the *head* field needs to cover two coherence actions: invalidation and downgrade. In particular, we have also considered two subtypes for the invalidation case[2] depending on whether the action involves a single owner (`M` in Figure 5.3b), or multiple sharers (`S` in Figure 5.3a). Note that, like in *Directory*, the home tile readily would identify both cases by means of different states at the L2 cache. Therefore, to cover all possible cases, this field contains two bits.

On the other hand, the *payload* field contains the block address of the *ACN* operation. This configuration is called `BlockAddress` and is outlined at the top of Figure 5.4b. While this is the most precise way to perform a coherence action, since the block address unequivocally identifies every copy in the L1 caches, we could consider a smaller number of bits in order to reduce propagation latency of the *ACN* messages. For example, every *ACN* message could contain a subset of the entire block address that would identify the cache entry where the block would be (i.e. the index). Nonetheless, the problem with this configuration, called `Index` and shown in the center of Figure 5.4b, would be when the L1 caches do not follow a direct-mapped scheme. For instance, in a *N-way* set-associative scheme, up to *N* cached blocks could be invalidated which may entail extra L1 cache misses that would degrade performance.

To alleviate such an issue, we propose two different optimizations. The first optimization will be referred to as `Index+ExtraBits` and is depicted at the bottom of the Figure 5.4b. Here, apart from including the index field, we could incorporate some extra bits taken from the block address to check against the same bits stored in the tag address of every L1 cache entry. So, the more extra bits included the lower probability of invalidating wrong cached blocks. As a second optimization, we propose to apply two types of filters at the L1 caches aimed at reducing the number of cached blocks affected by the imprecision that the *ACN* messages entail. On the one hand, the *state filter*, that prevents from invalidating cached blocks with irrelevant stable states: in downgrade or invalidation actions on owners' caches, stable states different than `M` or `E`; and for invalidations of multiple sharers, stable states other than `S`. On the other hand, the *sharing filter*, that would exclude all private blocks in the particular L1 cache entry from the coherence operations. Additionally, *ECONO* only maintains coherence for shared

[2]This will be necessary to support the *state filter* explained below.

blocks, which are the blocks that truly need coherence. In this chapter, to identify private blocks, we assume an OS-based classification similar to what is done in [13].

As to the *3-hop ECONO*, the *head* field needs to cover three coherence actions. First, invalidation, which is used to invalidate all shared copies in the system (e.g. for a write cache miss in Figure 5.3a). Second, forward-downgrade, which is used to downgrade the owner's copy and forward the data to the requesting core (for a read cache miss in Figure 5.3b). And third, forward-invalidate, where the owner forwards and invalidates its copy (for a write cache miss in Figure 5.3b). Consequently, in *3-hop ECONO*, the *head* requires two bits. Finally, for the *payload* field, we will only consider an *ACN* message in which the block address is used along with the requester's ID to identify the destination of the forward message. This configuration will be referred to as `ID+BlockAddress` and is shown in Figure 5.4c. The number of bits required by the ID is equal to the $\log_2(\#Tiles)$ (e.g. four bits for the 16-tile configuration assumed in this chapter). Besides, in this *ECONO* implementation, we will not make use of shorter and imprecise *ACN* messages. Imprecise forwards would be very difficult to manage because there may be forwards to wrong owners, forwards to the correct owner on wrong blocks, or even data responses to wrong requesters from correct or wrong owners.

5.3.3 Physical Implementation

ECONO protocol has been designed and implemented considering the target many-core tiled-based CMP architecture presented in Section 2.1. In short, this system is composed of a two-level hierarchy including one private level of instruction and data caches, and a logically-shared physically-distributed L2 cache. Moreover, a 2D-mesh layout is used for the CMP's interconnect. Figure 5.5 illustrates this system including some of the components required by *ECONO* that we described below.

Differently from the two other proposals presented in this thesis, *GBarrier* (Chapter 3) and *GLock* (Chapter 4), in this chapter we will consider an implementation of *ECONO* based on the full-custom *G-Lines* technology. An experimental analysis using the other type of technology considered in this thesis, the *Standard* one, revealed that this is not efficient enough to provide neither lightweight infrastructure nor very fast broadcast operations for our coherence protocol. Not surprisingly, other recent broadcast-based coherence protocols for many-core CMPs [44, 154] (further details in Section 5.6) come to the same conclusion,

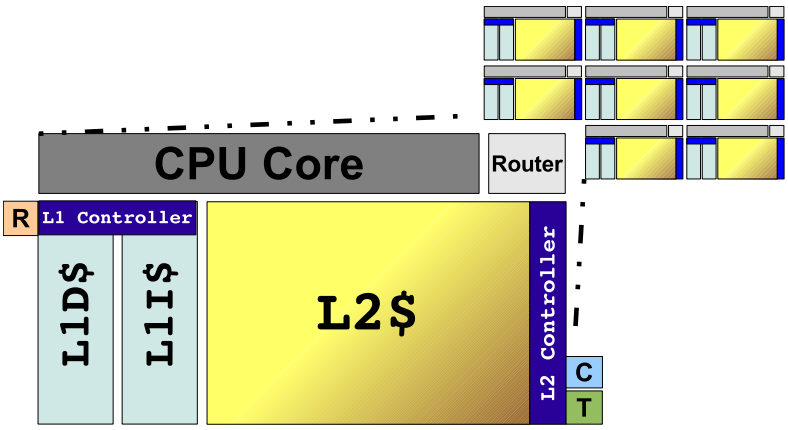


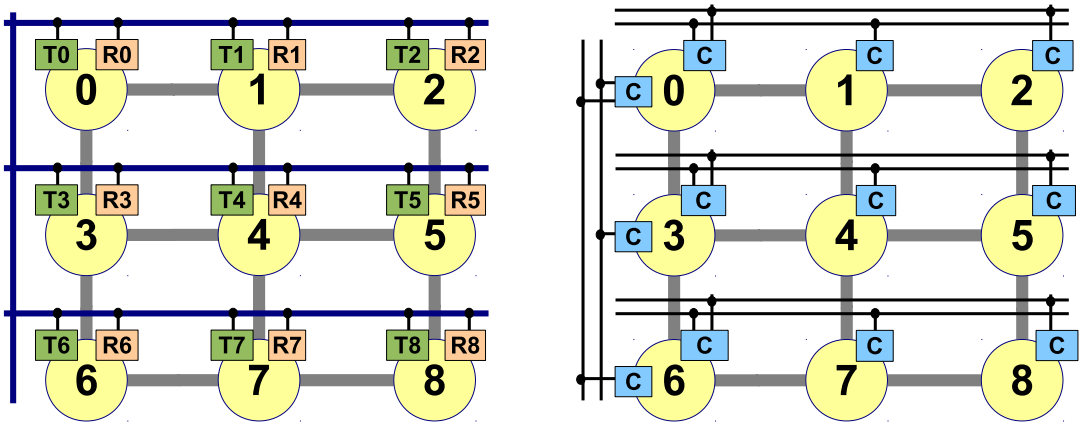
Figure 5.5: Extra HW resources per Tile: `C`, `T` and `R` controllers.

though they discard the *Standard* technology in favor of another state-of-the-art technology based on new advances in nanophotonic transmissions [123].

The physical implementation for the *ECONO* protocol relies on very simple hardware extensions to the CMP system in terms of a few controllers per tile and the use of a very lightweight dedicated on-chip network to convey the *ACN* messages. Moreover, to ensure atomicity in the transmission of such messages, another lightweight dedicated on-chip network is devoted based on the *GLock* infrastructure presented in Chapter 4. As we will explain, these hardware extensions modify neither the processor core nor the main interconnection network at all, and they require enabling a couple of interfaces between cache controllers and the *ECONO* architecture to communicate with each other.

5.3.3.1 Controllers

Figure 5.5 shows the three types of controllers per tile required by our proposal: `T`, `R` and `C` controllers. These controllers can be classified into two groups depending on the functionality they provide. The first group consists of the `T` and `R` controllers, that will operate as transmitters and receivers of the *ACN* messages over *ECONO*'s network, respectively. Since the home tile broadcasts these messages to all L1 caches, `T` is implemented in the L2 controllers, whereas `R` is implemented in the L1 controllers. The second group refers to the `C` controllers that will be devoted to guarantee the atomic transmission of the *ACN* messages. More specifically, `C` will be in charge of requesting, acquiring and releasing the *ECONO*'s network ownership to transfer the message in mutual exclusion. Due



(a) *GecNetwork* (b) *GLock*

Figure 5.6: Dedicated networks required for a 9-tile CMP.

to the fact that this task is performed at the home banks, as we can observe in Figure 5.5, the `C` controllers are implemented as part of the L2 cache controllers.

5.3.3.2 Dedicated On-Chip Networks

The two on-chip networks required by our coherence protocol have been developed considering the most efficient *G-Lines* technology. In this way, both infrastructures must be composed of long 1-bit wires (i.e. a *G-Line*) to interconnect all the different controllers previously described. To simplify the design, we decided to employ a 2D-mesh topology for both networks because it is the topology used in the main CMP's interconnect.

Figure 5.6 illustrates both networks for a 3×3-tile CMP system, where every circle represents a tile, thick gray lines constitute the main 2D-mesh interconnect, and finer lines are our networks with their respective controllers depicted as boxes. As we can observe, each of the two groups of controllers presented above operates over a different on-chip network: the *G-Line*-based express coherence network (*GecNetwork*), for `T` and `R` controllers; and the *GLock*'s network, for `C`.

Regarding the *GecNetwork* (see Figure 5.6a), it interconnects all `T` and `R` controllers and enables broadcasts of the *ACN* messages by using horizontal and vertical lines following a three-phase scheme. First, a particular `T` controller writes a message into its horizontal line. Then, the horizontal line in which `T` is attached to broadcasts the message to all the `R` controllers attached to the same line. Second, a vertical line is devoted to broadcast this message to the remaining

horizontal lines. And third, these horizontal lines broadcast the message to the rest of R controllers. Notice that, acquiring network ownership guarantees that just a single home tile will be able to transmit a message, what prevents from any interference among *ACNs* and establishes a deterministic way to estimate propagation latency.

The *GLock* architecture shown in Figure 5.6b will guarantee exclusive access to the *GecNetwork* for a particular home tile. Since there may be multiple home tiles competing for network ownership, it would be necessary to use a fair and efficient mechanism to resolve possible scenarios under high degree of contention. Due to its similarity to the problem of efficiently managing highly-contended locks in parallel applications, we adapt our *GLock* proposal presented in Chapter 4 to manage contention at L2-bank level (i.e. every home tile). This adaptation was straightforward because it simply requires a change in the elements that activate the *GLock*: L2 banks rather than processor cores. The hardware resources of the *GLock* infrastructure will be taken into account when analyzing the total hardware cost of *ECONO*'s infrastructure in Section 5.4.2.

5.4 Performance Implications

In this section, we analyze *ECONO* to determine its potential impact on execution time and required hardware resources considering area overhead, power dissipation and scalability.

5.4.1 *ACN* Latency

The performance of *ECONO* is highly dependent on the efficiency of the atomic broadcast operations to transmit the *ACN* messages it triggers in order to ensure coherence. As explained in Section 5.3.1, three different steps are required to transmit an *ACN* message: the *GecNetwork* acquisition; the *ACN* message transfer; and the *GecNetwork* release. Due to the fact that the first and last steps are accomplished by the *GLock* mechanism, according to the raw performance statistics reported in Section 4.2.5, the *GLock* takes: 4 or 2 cycles (worst or best case) for the first step; and 1 cycle for the last one.

Regarding the second step, Figure 5.7 depicts the transfer delays (y-axis) depending on message sizes (x-axis) and number of *G-Lines* used in the *GecNetwork* (bandwidths ranging from 1 to 7 bits per clock cycle). The total number of bits required by the different message sizes takes into account the simulated

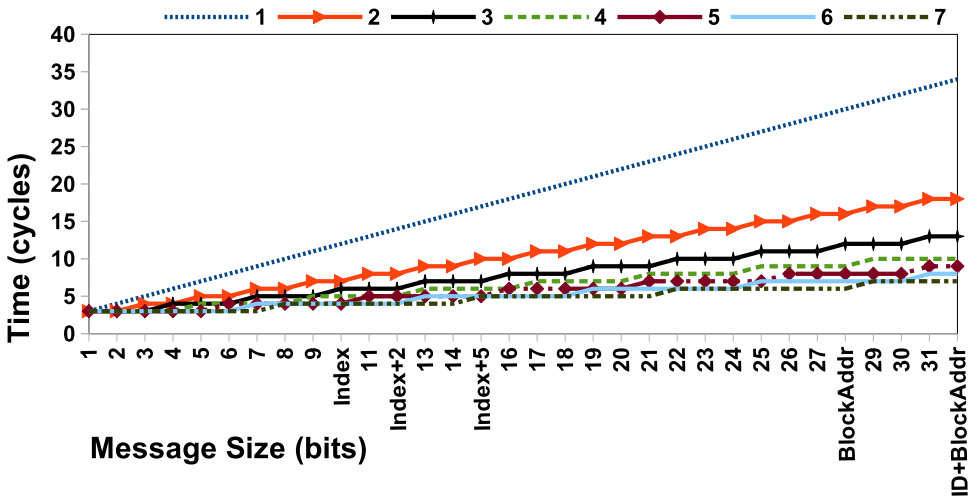


Figure 5.7: Propagation delays for *ACN* messages using different *G-Lines* for the *GecNetwork*.

parameters for the CMP system detailed in Table 5.2. The timing values plotted in the figure come from the 3-phase broadcast scheme discussed in Section 5.3.3.2 to transmit an *ACN* message. To accelerate the process, we implement a pipeline strategy by overlapping transmissions of different parts of the message along with the three different phases of the broadcast. In this way, the estimation of propagation delays was computed by means of Equation 5.1, where `MessageSize` corresponds to the number of bits to transfer, and `G-Lines` stems from the number of 1-bit-width *G-Lines* employed.

$$PropagationDelay = \left\lceil \frac{\texttt{MessageSize}}{\texttt{G-Lines}} \right\rceil + 2 \tag{5.1}$$

From Figure 5.7, we can derive two important conclusions. First, as expected, the greater the message size is, the longer the latency is. Second, there exists a tradeoff between bandwidth and on-chip area overhead of the *GecNetwork*. For instance, the 7-*G-Line GecNetwork* achieves the lowest propagation delays but it involves the greatest area overhead. As we can observe, the use of three to seven *G-Lines* achieves almost negligible relative improvements (5 clock cycles in the worst case), but it obtains important area savings when choosing the smaller number of *G-Lines*. Consequently, we evaluate our *ECONO* mechanism considering a 3-*G-Line GecNetwork* infrastructure.

5.4.2 Hardware Resources

Table 5.1: Hardware cost of *ECONO* architecture for a 2D-mesh CMP layout.

1-bit *GecNetwork*	*G-Lines*	$\sqrt{C}+1$
	Controllers: Ts+Cs	$C+C$
GLock	*G-Lines*	$C-1$
	Controllers	$\sqrt{C}+C$

The hardware cost of our proposal for the target 2D-mesh CMP layout considered in this thesis comes as consequence of these two components: the *GecNetwork* (Figure 5.6a) and the *GLock* (Figure 5.6b). Table 5.1 details the number of *G-Lines* and *Controllers* that both components require. In particular, assuming 1-bit *G-Lines*, and being C the total number of cores or tiles, the *GecNetwork* needs a set of $\sqrt{C}+1$ *G-Lines* (see the four *G-Lines* in Figure 5.6a) as well as C `T` controllers plus C `R` controllers (see the eighteen controllers shown in Figure 5.6a). In general, for a *P-G-Line GecNetwork* the latter numbers must be multiplied by a factor of P. Besides, from the analysis carried out in Section 4.2.5, the set of *G-Lines* and *Controllers* for the *GLock* would be $C-1$ and $\sqrt{C}+C$, respectively.

To provide some insight into the magnitude of the hardware cost exposed, we consider the 16-tile CMP later evaluated in Section 5.5. Therefore, the total number of *G-Lines* required for *ECONO* would be equal to 30 (15 for 3-*G-Line GecNetwork* plus 15 for *GLock* architecture), whereas the total number of controllers would be equal to 116 (96 and 20, respectively). As reported in [142], a 392-*G-Line* network shows very small area implications, so we can assume that our proposal introduces negligible area overhead.

5.4.3 Power Dissipation

Since the *ACN* messages are always broadcast, the total power of our proposal is dominated by the dissipation of the *GecNetwork*. The *GLock* is much more energy efficient because it only requires the transmission of a single 1-bit message towards the next home tile that acquires the *GecNetwork* ownership. However, we will take into account a worst-case scenario in which all home tiles request the *GecNetwork* ownership at the same time. For the estimation of the power dissipated by *ECONO*, we employ the power dissipation parameters for a 65-nm CMOS process simulated in [142]: 0.6 mW per `T` controller; and 0.4 mW per `R`

controller. Moreover, according to [142] no static power is dissipated using the *G-Line* circuitry.

On the one hand, for the power dissipated by the *GLock*, we must deal with the worst-case scenario described above. In particular, there could be up to three local controllers per row requesting lock ownership (i.e. 12 transmitters for the whole CMP) and four secondary lock managers, where each one keeps track of the three signals transmitted by its three local controllers (i.e. 12 receivers for the whole CMP). Thereby, the total power estimated is equal to 12 mW (12×0.6 + 12×0.4).

On the other hand, regarding the power dissipation derived from the *GecNetwork*, we firstly determine the maximum number of transmitters and receivers operating at once when the *ACN* message is being transmitted. Therefore, considering a 1-*G-Line GecNetwork* and the pipeline-based three-phase strategy to broadcast every message explained in Section 5.4.1, there will be a maximum number of 5 `T` controllers and 19 `R` controllers in a given clock cycle. More specifically: 1 `T` controller and $3 + 1r$ `R` controllers in the horizontal *G-Line*; $1r$ `T` controller and $3r$ `R` controllers in the vertical *G-Line*; and $3r$ `T` controllers and 12 `R` controllers for the rest of horizontal *G-Lines* (the r suffix stems from the `R` and `T` controllers that would work as repeaters in the vertical *G-Line* outlined in Figure 5.6a). Therefore, the total power estimated for the *GecNetwork* will be 10.6 mW per clock cycle ($5 \times 0.6 + 19 \times 0.4$). In particular 31.8 mW, if we consider the 3-*G-Line GecNetwork* required by *ECONO*.

In consequence, the total power dissipation required by *ECONO* in the worst-case scenario will be equal to 43.8 mW (12 mW for the *GLock* and 31.8 mW dissipated by the *GecNetwork*). Utilizing CACTI [52], the magnitude of this dissipation is approximately one-7th of the power dissipated per read port in the L1 caches simulated in this chapter (see Table 5.2).

5.4.4 Scalability

Both the *GecNetwork* and the *GLock* infrastructures are built upon a *G-Line*-based fabric. As in [142], every *G-Line* can support up to seven controllers resulting in a CMP configuration with up to 7×7 cores. We would like to point out that our mechanism is not restricted to that CMP layout and could be easily extended to support a higher number of cores. In particular, we could adopt a similar strategy to that discussed for *GBarrier* and *GLock* in Chapters 3 and 4, respectively. More specifically, we could increase the scalability of *ECONO* by using either longer-latency *G-Lines*, or different groups of *G-Line*-based networks linked together through additional *G-Lines* following a hierarchical layout.

Table 5.2: CMP baseline configuration.

Number of cores	16
Core	3 GHz, in-order, single-issue
Cache line size	64 Bytes
L1 I/D-Cache	32 KB, 4-way, 2 cycles
L2 Cache Bank	512 KB, 8-way, 12+4 cycles
Memory access time	250 cycles
Network configuration	2D-mesh
Data message size	72 bytes
Control message sizes	8 bytes
Network bandwidth	48 GB/s
Flit size	16 bytes
Link bandwidth	1 flit/cycle

Alternatively, to also achieve high scalability, our proposal could also be easily implemented assuming the leading-edge nanophotonic technology [123].

5.5 Evaluation

In this section, we firstly expose the experimental setup utilized in this chapter in order to evaluate the performance benefits derived from our *ECONO* mechanism. Next, in the evaluation part, we conduct a number of experiments to determine the best-performing configurations for *ECONO*, and from the resulting settings, we carry out a performance comparison against the two contemporary coherence protocols considered in this chapter: *Hammer* and *Directory*.

5.5.1 Experimental Setup

The evaluation of *ECONO* has been carried out by using the full-system simulation tool explained in Section 2.2.2: Virtutech Simics [118] (running Solaris 10) extended with Wisconsin GEMS toolset [98], and we also use its GARNET [107] component to obtain a precise modeling of the interconnection network. Table 5.2 summarizes the values of the main configurable parameters assumed in this chapter. In short, we simulate a 16-tile CMP with a 2D-mesh topology and a two-level inclusive hierarchy.

Table 5.3: Benchmarks and input sizes.

Benchmarks	**Input Size**
SPLASH-2 (5)	
Barnes	8192 bodies, 4 time steps
FFT	64K complex doubles
Ocean	258×258 ocean
Radix	1,048,576 radix
Raytrace-opt	Teapot
Scientific Applications (3)	
EM3D	38,400, degree 2, 15%, 50 time steps
Tomcatv	256 points, 5 time steps
Unstructured	Mesh.2K, 5 time steps
PARSEC (1)	
Swaptions	simmedium

As benchmarks, we use nine multi-threaded applications: five from the SPLASH-2 benchmark suite [128], one from the PARSEC benchmark suite [18], and three scientific applications. Table 5.3 shows them and their respective problem sizes. These applications were chosen because they exhibit different communication patterns ranging from small coherence activity like in Barnes to high activity like in Swaptions, as we will see in Section 5.5.3. All experimental results reported in this chapter are for the parallel phase of the benchmarks under study.

To quantify the performance benefits derived from our proposal we conduct two sorts of experiments. First, we determine the best-performing configurations for *ECONO* according to the extensions presented in Section 5.3.2. And second, the resulting *ECONO* implementations are compared against the two contemporary coherence protocols considered in this chapter: *Hammer* [2] and *Directory* [88]. To ensure a fair comparison in terms of performance, all protocols share a common specification that consists of write-invalidate policy, MESI state machines for the first-level caches, and inclusive cache hierarchies with write-back caches and an invalidation-on-eviction policy for L2 replacements. In consequence, the *Hammer* implementation evaluated in this chapter is an optimized version of [2] because it has precise knowledge about the memory blocks that are cached. Note that, otherwise, broadcasts would always be sent to both on-chip caches and

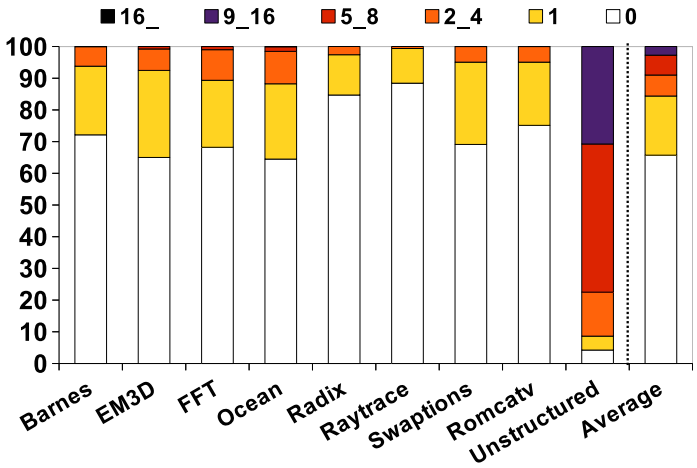
memory in order to obtain the requested data. This would generate extra traffic into the interconnect when data is off-chip, and important performance degradation waiting for off-chip responses when they are no needed. Moreover, *Directory* has been implemented by using upgrade requests. This entails an optimization in terms of network traffic because an upgrade is used to request write permission for a cached block with only read permission. If the requester still remains as a sharer at home's coherence information, an acknowledgement message from the latter would be sent instead of the home's copy of the block, thereby saving network traffic (i.e. 8 vs. 72 bytes according to Table 5.2). Note that, the rest of the protocols evaluated in this chapter do not store any coherence information, and therefore, this optimization cannot be applied.

5.5.2 Characterization of the *ECONO* Protocol

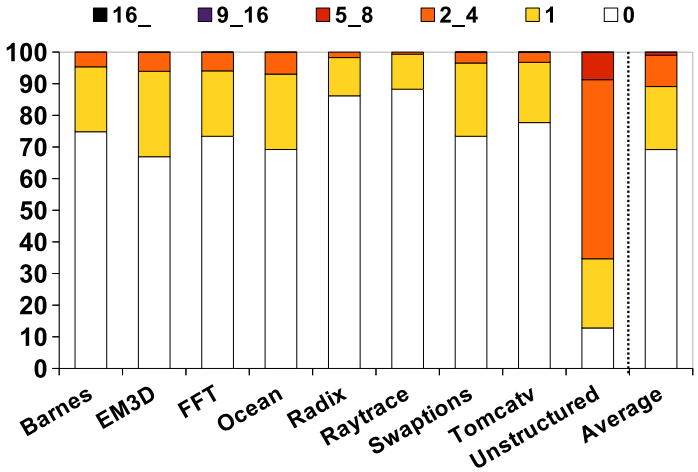
The different extensions to the baseline *ECONO* implementation presented in Section 5.3.2 are evaluated in order to find out the best-performing configuration.

5.5.2.1 Number of *GecNetworks*

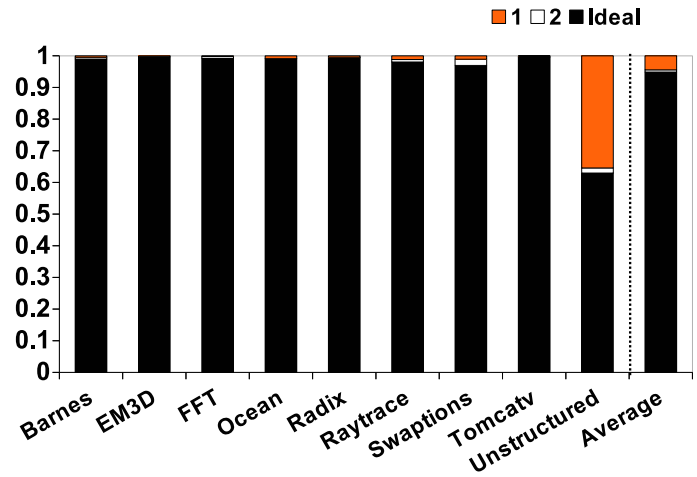
From the analytical study carried out in Section 5.4.1, we conclude that the *GecNetwork* must be comprised of a 3-*G-Line* infrastructure since a greater number of *G-Lines* leads to marginal improvements. Now, we investigate if this configuration is enough to keep up with high performance or, on the contrary, it will require more *GecNetworks* in order to solve possible problems of contention when dealing with real applications. Remember that contention is due to mutual exclusion required by the transmission of an *ACN* message. For the experiment, we consider the worst-case scenario in which the messages contain the largest *payload* in the *4-hop ECONO*: the `BlockAddress`. Figures 5.8a and 5.8b show the effect of contention on execution time for the set of benchmarks under study, when dealing with one or two *GecNetworks*, respectively. Besides, each bar is broken down into six categories depending on the concurrent home tiles that are waiting for the *GecNetwork* acquisition: `0` when the network is free; `1` when there is one home tile which either is gaining access to the network, or its *ACN* message is still being sent; and for example `2_4`, where there are from two to four home tiles waiting for sending the corresponding message. Note that, despite considering a 16-tile CMP, it is possible that more than 16 concurrent petitions may be in progress because once a particular home tile acquires the ownership and sends the *ACN* message, in the interim, other served requests could require



(a) Contention for a single *GecNetwork*.



(b) Contention for two *GecNetworks*.



(c) Normalized execution times.

Figure 5.8: Performance depending on the number of *GecNetworks*.

another *ACN* operation. Nevertheless, this situation has not been reflected in the experiments because of being a very infrequent event.

As we can observe in Figure 5.8a, 65% on average of the accesses encounter that the *GecNetwork* is free, and considering up to four petitioners for low contention an average of 90%. The exception is Unstructured that presents a high degree of contention since for 30% of the accesses more than nine petitioners are waiting for sending their *ACN* message. Figure 5.8b discloses interesting results when considering two *GecNetworks* because the degree of contention is reduced considerably. For example, in Unstructured the percentage of accesses is now marginal when considering more than nine petitioners.

The previous results were analyzed to determine the impact on execution time when dealing with different numbers of *GecNetworks*. In consequence, Figure 5.8c shows the normalized execution times that are obtained for the set of benchmarks under study when dealing with one, two and infinite (`Ideal`) networks. The execution times are normalized with respect to those obtained for the worst case of having a single *GecNetwork*. Moreover, each bar has been plotted by overlapping the normalized execution times of every case (`1`, `2` and `Ideal`). This overlap is depicted by highlighting the shorter times firstly. For instance, the bar for Unstructured shows: 0.63 for `Ideal` (the shortest time), next up a fraction of 0.02 for `2` (actually 0.65), and 0.35 for `1` (actually 1). As we can observe, the reduced contention derived from using two networks as aforementioned translates into important performance benefits in execution time. Note that, the highest contended application, Unstructured, presents a reduction of 35% when considering two *GecNetworks*. Another interesting result is that, even an infinite number of *GecNetworks* reports almost the same improvements.

As a conclusion of this first set of experiments, we can affirm that two *GecNetworks* is enough to achieve the highest performance for the *4-hop ECONO*. This study was also conducted for the *3-hop ECONO* and the very same conclusion was obtained.

5.5.2.2 Type of *ACN* messages

As discussed in Section 5.3.2, we proposed imprecise *ACN* messages to shorten propagation delays but at the expense of affecting more cached blocks than necessary (*wrong blocks*). The next set of experiments are aimed at quantifying the performance degradation due to the presence of *wrong blocks*. To reduce the number of *wrong blocks*, we also proposed to make use of two kinds of filters: the *state* and *sharing* filters. In this section, we assume that both filters are activated

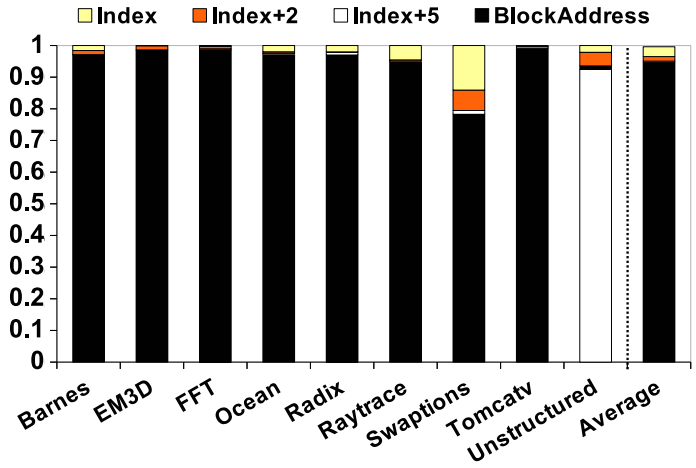
and later on, in Section 5.5.2.3, we will complete the study by quantifying the performance penalties derived from not applying either a single or both of such filters.

To conduct this exploration, we consider the three types of imprecise *ACN* messages discussed in Section 5.3.2: `BlockAddress`, `Index` and `Index+ExtraBits`. In addition, we focus on the *4-hop ECONO* implementation since imprecise *ACN* messages for the *3-hop* implementation falls into unmanageable complexity (further details in Section 5.3.2). Finally, as concluded in Section 5.5.2.1, the results are obtained relying on two 3-*G-Line GecNetworks*.

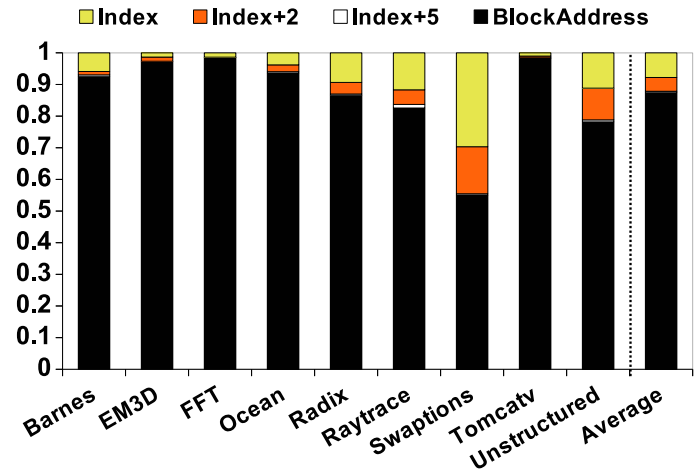
In Figure 5.9a, we show the performance implications on execution time for the applications under study. For that, we normalize execution times with respect to *Index* because it presents the highest imprecision and then, the highest performance degradation. Remember that this message only stores the bits required to identify the particular set in the L1 caches where the requested block would be. Therefore, for the simulated 4-way set-associative caches, up to three blocks per cache may be *wrong blocks*. It is worth noticing that imprecise information reports shorter *ACN* latencies what may lead to overcome the precise configuration of `BlockAddress`. As for previous Figure 5.8c, we plot overlapped results for each bar.

Comparing `Index` to the precise `BlockAddress` configuration, Figure 5.9a shows an average performance improvement of 4% in favor of the second option, though in Swaptions the performance gap increases up to 22%. This motivates the use of less imprecise schemes such as `Index+ExtraBits`. For that, we study `Index+2` and `Index+5` configurations. The reason why we decided to use two and five extra bits is that these numbers maximize the information transmitted in every *ACN* message, and their message sizes are a multiple of three (the number of *G-Lines* per *GecNetwork*). As we can see in the figure, lower imprecision leads to lower values in execution time. In consequence, `Index+5` is able to achieve similar results to the precise `BlockAddress` and is even slightly better for Unstructured (the white fraction of the bar is in front of the black one). Note that this application presents the highest contention (Figure 5.8a) and takes benefit from a lower propagation delay to outperform the precise configuration. More specifically, `Index+5` involves *ACN* messages that are 46% shorter (15 vs 28 bits), which implies a propagation delay which is 41% lower (7 vs 12 clock cycles) as shown in Figure 5.7. In addition, we observed that a greater number of extra bits does not report better results.

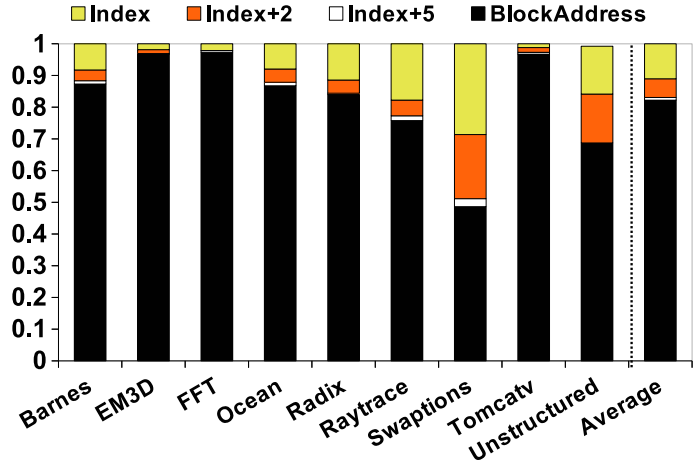
In Figure 5.9b we illustrate the performance implications on network traffic that results from using imprecise *ACNs*. The results shown in this figure are



(a) Normalized execution time.



(b) Normalized network traffic.



(c) Normalized L1 cache misses.

Figure 5.9: Performance characterization for different types of *ACN* messages.

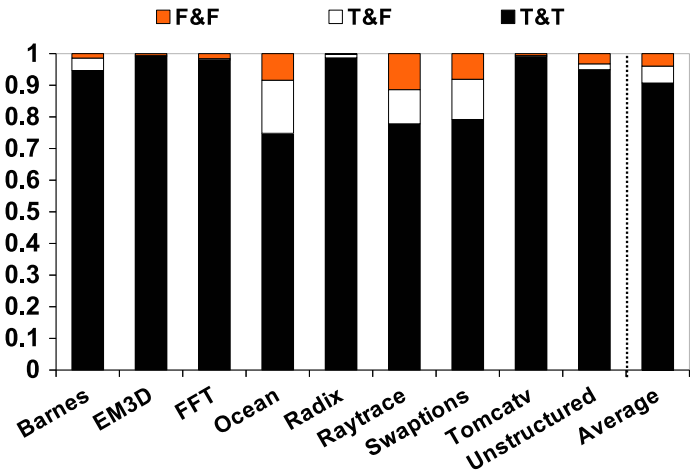


Figure 5.10: Effect on L1 cache misses depending on the use of the *state* and *sharing* filters.

derived from the extra L1 cache misses when *wrong blocks* are affected by the *ACN* operations (i.e. more requests and data movements). We depict overlapped results and normalized with respect to `Index`. As we can observe, the precise `BlockAddress` reports the best results and obtains improvements of 13%, on average. Moreover, `Index+5` reports roughly the same results than the precise configuration. Finally, we quantify in Figure 5.9c the extra L1 cache misses derived from *wrong blocks*. As we can observe, the magnitude of these results is directly related to the extents of the results in network traffic previously reported.

As a conclusion, we can say that the `Index+5` configuration will be the preferred choice because it behaves as the precise `BlockAddress` as shown on average results, but it may get to the point of outperforming the latter because of the shorter propagation delays that it implies. Therefore, we choose `Index+5` for the *4-hop ECONO* implementation evaluated later on.

5.5.2.3 Application of Filters

The *state* and *sharing* filters have been proposed to impede invalidating more blocks than necessary (*wrong blocks*) when using imprecise *ACN* messages, thereby saving extra L1 cache misses and improving execution time. To quantify the benefits from applying the filters, Figure 5.10 shows the extra L1 cache misses when the filters are or not activated using the `Index+2` configuration as a case study. For that, we normalize L1 cache misses with respect to the worst configuration in

which both filters are disabled. Moreover, the legend of the figure shows different configurations for `X&Y` meaning that either *state* (`X`) or *sharing* (`Y`) filters are on (`T`) or off (`F`). Notice that, we plot overlapped results for each bar.

As we can observe in the figure, a maximum improvement of 25% (10% on average) is obtained when applying both filters (the `T&T` configuration), and they separately represent roughly half the total benefit achieved. Therefore, as a conclusion of this section we can say that, due to the significant reduction in L1 cache misses that the applicability of both filters achieves, the best-performing configuration for *4-hop ECONO* using imprecise *ACN* messages evaluated in next section will incorporate both filters.

5.5.3 Performance Results

In this section we compare the best-performing configurations for *4-hop* and *3-hop ECONO* against *Hammer* and *Directory* in terms of execution time and network traffic for the benchmarks under study.

Before starting with the evaluation part, we depict in Figure 5.11 a characterization of the different coherence actions performed by the *Directory* protocol in order to help understand where improvements come from in the next two sections. We choose *Directory* because this protocol distinguishes a more diverse set of coherence actions to perform, what provides more information for the performance comparison (further details in Section 5.2.2).

As we can observe in the figure, each bar is broken down into four categories depending on the percentage of coherence actions devoted to: forward messages, that comprehend all coherence messages transmitted from the home tiles to the owners in order to recover the single valid copy of the requested block (`Fwd` in the figure); upgrade actions, that result from granting write permission to a requesting core that has a read-only copy of a cached block (i.e. it implies invalidation of sharers and an acknowledgement sent to the requester, or `Inv&Ack`); coherence activity devoted to grant write permission to a requester that does not have a block that is shared at the home tile (i.e. it involves invalidation of sharers and delivery of the home's block copy to the requester, or `Inv&Data`); and finally, those actions that only require sending the requested data to the requesting core (i.e. `Data`).

A preliminary analysis of the results shown in Figure 5.11 reveals that we cannot expect to find significant performance differences among the four protocols for benchmarks that are dominated by coherence actions belonging to the `Data`

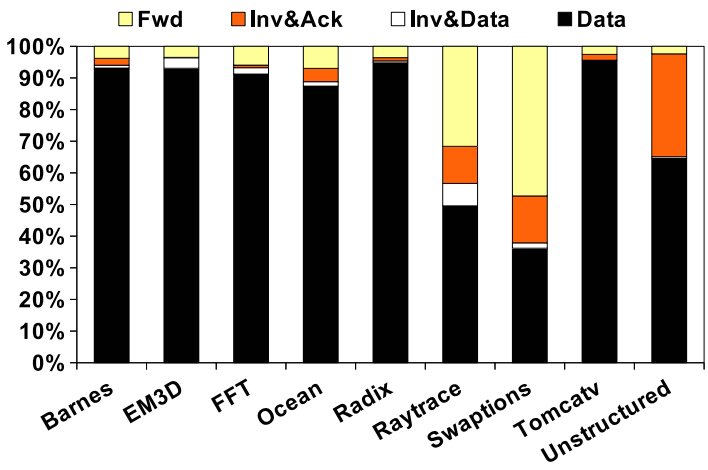


Figure 5.11: Characterization of coherence activity in the *Directory* protocol.

category (i.e. all but Raytrace, Swaptions and Unstructured), because in such benchmarks all the protocols would behave in the same way.

5.5.3.1 Execution Time

Figure 5.12 shows the normalized execution times with respect to those obtained when *Hammer* is considered. As expected, *Directory* achieves important reductions in execution time (14% on average) because of using precise coherence information which entails less coherence messages to be transmitted (e.g. Invalidations) and to wait for (i.e. Acknowledgements). More specifically, the magnitude of these savings depends on the number of coherence operations optimized by *Directory* (i.e. those that fall into `Fwd`, `Inv&Ack` and `Inv&Data` categories in Figure 5.11). This is the reason why negligible performance improvements are obtained in all but Raytrace, Swaptions and Unstructured benchmarks as alluded to above in our preliminary analysis. Besides, the highest improvements are achieved in Swaptions because in this benchmark roughly 50% of the coherence actions involve a single point-to-point forward message (see `Fwd` category in Figure 5.11). Conversely, in *Hammer*, broadcast messages are employed, which increases execution time mainly for two reasons: first, due to higher contention at the main interconnect; and second, more time spent at requesting cores waiting for the corresponding acknowledgement messages.

Regarding our *ECONO* implementations, by relying on a single *ACN* message transmitted in broadcast over a dedicated very fast *G-Line*-based on-chip network,

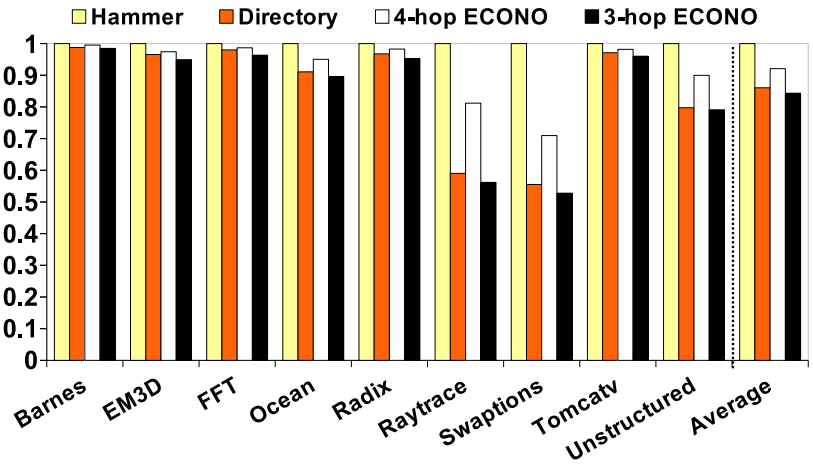


Figure 5.12: Normalized execution times.

both implementations outperform the *Hammer* protocol. Nevertheless, *4-hop ECONO* requires one more hop when data recovery is necessary as we discussed in Figure 5.3b, and then, it cannot outperform *Directory*, that manages this situation much more efficiently than *Hammer* does as follows. First, similar to *ECONO*, *Directory* would require a single coherence message sent to the particular owner that eventually would be in charge of sending the data to the requester. Second, similar to *ECONO*, the requester would not spend any time waiting for acknowledgement messages. In particular, Raytrace and Swaptions report the higher performance gap between *4-hop ECONO* and *Directory* because, as shown in Figure 5.11, these benchmarks have a fraction of approximately 30% and 50% devoted to forward messages, respectively (see the `Fwd` category). Finally, when analyzing the execution times reported by *3-hop ECONO*, we can observe that this new implementation takes benefit from using less number of hops thereby even achieving slightly better results than *Directory* (2% on average).

5.5.3.2 Network Traffic

Figure 5.13 shows the total network traffic depending on the implementations discussed above, normalized with respect to *Hammer*. In particular, each bar plots the number of bytes transmitted through the interconnection network (the total number of bytes transmitted by all the switches of the interconnect). As expected, *Directory* outperforms *Hammer* (27% on average) because of using precise information of cached blocks that removes all unnecessary coherence messages

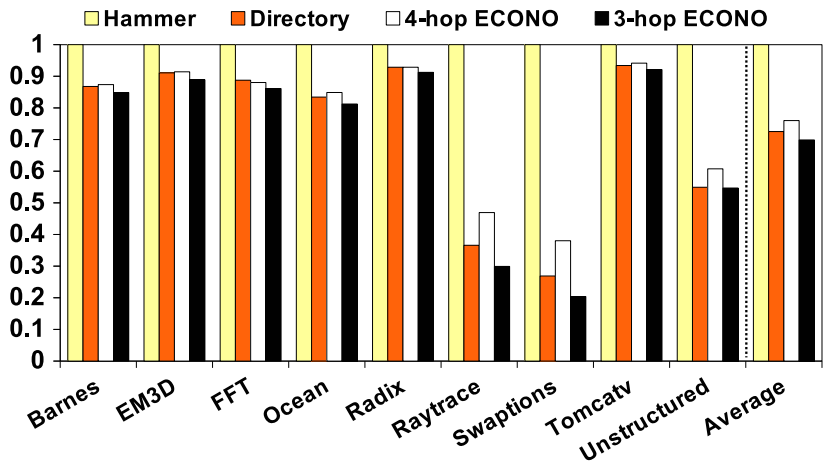


Figure 5.13: Normalized network traffic.

from the interconnect. According to our characterization in Figure 5.11, these improvements come as consequence of the aggregate fraction of `Fwd`, `Inv&Ack` and `Inv&Data` categories, since for such coherence actions *Directory* always performs better than *Hammer*: `Fwd` and `Inv&Data` would require broadcast operations; and `Inv&Ack` would be implemented as `Inv&Data` since upgrade operations are not implemented in *Hammer* because of lack of sharing information at home tiles.

Regarding the *ECONO* implementations, recall that our protocol removes an important amount of coherence-related traffic from the main CMP's network (i.e. the home tiles transmit *ACN* messages over *ECONO*'s special network rather than invalidation or forward messages over the main interconnect, and no acknowledgements are necessary), so that it must lead to significant reductions in network traffic.

Considering the *4-hop ECONO* implementation, we can observe that it does not outperform *Directory* (degradation of 3% on average). The reason is that requiring indirection to the home tiles for data recovery (see Figure 5.3b) implies more hops and then, more messages are injected into the main interconnect to convey data blocks to the requesting cores. Note that, according to the simulated parameters shown in Table 5.2, control messages are one-9th of the size of data messages, and then injecting more data messages to reach the requester's cache nullifies the benefits of removing all coherence requests and acknowledgements for almost all the benchmarks.

As to the *3-hop ECONO* protocol, due to the fact that forwarding operations are enabled important reductions are reported. As we can observe in Figure 5.13, this implementation saves network traffic by 3% compared to *Directory*. In particular, applications such as Swaptions or Raytrace, that present frequent invalidation of sharers and forward operations (see `Inv&Data`, `Inv&Ack` and `Fwd` categories in Figure 5.11), report the highest improvements. Moreover, Unstructured reports similar results to *Directory* because, in this benchmark almost 40% of the coherence activity observed in *Directory* is optimized by using upgrade petitions that replace data responses with the shorter acknowledgement messages (see `Inv&Ack` category in Figure 5.11).

As a final observation, from the reductions in execution time and network traffic that *ECONO* entails, and due to the negligible extra power dissipation required by the required infrastructure estimated in Section 5.4.3, we can also affirm that the *4-hop ECONO* is more energy efficient than *Hammer*, and that the *3-hop ECONO* implementation is the most energy efficient design.

5.6 Related Work

The maintenance of cache coherence in shared-memory parallel systems has been a first-order design issue for many years [91]. Nowadays, many-core CMPs demand new coherence protocols that provide the required efficiency and scalability to successfully harness their peak computational power as core count increases and technology scaling improves. In this section, we describe the main modern coherence protocols developed to date in the context of shared-memory many-core CMPs. Typically there are two main categories of coherence protocols: snooping-based protocols and directory-based protocols.

Snooping-based coherence protocols have been devised to maintain coherence in a simple way by broadcasting coherence requests to all cores in the system. Although these protocols work well in small-scale systems, their performance is highly compromised beyond a handful of cores, due to their prohibitive requirements of network bandwidth. There are three main proposals aimed at optimizing this category of protocols relying on snoop filters. Destination filters [147] remove probes at the receiver to reduce tag lookups but do not reduce messaging overhead. Source filters [60] eliminate network messages for unshared data, but still require broadcasts for lines with few sharers. And in-network filters [106], that remove probes at routing points in the network so that probes

only reach destination caches that are sharers of the memory block. However, this technique incurs the cost of additional complexity in the network.

In comparison to our proposal, *ECONO* manages coherence traffic much more efficiently than snooping-based protocols because it is not bandwidth intensive mainly for two reasons. First, by operating in a similar way to directory-based protocols, the requesting cores send an unicast message towards the home tile for every cache miss, hence not broadcasting this kind of messages. And second, as discussed in previous sections, the use of a very efficient *G-Line*-based lightweight infrastructure to broadcast *ACN* messages removes a considerable amount of coherence-related traffic from the main interconnect.

The second category of coherence protocols is based on storing an on-chip directory structure with coherence information about block sharers (e.g. a full-map sharing code [4]), so that coherence actions affect just the necessary caches, thereby leading to important traffic and energy savings. Nonetheless, directory-based protocols devote on-chip area overhead and energy consumption to store and search information about block sharers what may jeopardize efficiency and scalability of future many-core CMPs. That is the reason why there are a number of proposals that attempt to minimize the directory overhead. For instance, Tagless directory [84] uses an implicit, conservative representation of sharing information based on a grid of small bloom filters. The bloom filters concisely summarize the tags for each set, representing a superset of all of the block sharers (i.e. false positives occur), hence they completely eliminates the associative search on lookups of a conventional directory. Moreover, SPATL [51] adopts the same Tagless directory's approach of compressing the sharing information using bloom filters. SPATL is also based on identifying the sharing patterns of memory blocks that appear in the parallel applications, that is, the subset of cores that access to a memory block. This protocol holds every of the unique patterns found in a table's entry where multiple bloom filters with the same pattern point to. This leads to even further compression of the sharing information. Finally, SCD [36] exploits the insight that directories need to track a fixed number of sharers, not addresses, by representing sharer sets with a variable number of tags: lines with one or few sharers use a single tag, while widely shared lines use additional tags. SCD also leverages recent highly-associative caches (Zcaches [35]) to obtain a very efficient replacement process.

It is important to point out that unlike directory-based protocols, our *ECONO* protocol does not devote any storage overhead to track information about sharers, thereby saving on-chip area and energy. Moreover, the use of a dedicated on-chip network removes an important amount of coherence-related traffic from the main

interconnection network. This translates into higher efficiency in network traffic than directory-based protocols, as we concluded in the experiments conducted in Section 5.5.3.2.

Additionally, we can also identify another category of coherence protocols that results from removing the directory in the latter directory-based category and resorting to broadcasts. That is, we refer to broadcast-based protocols like the *Hammer* protocol discussed in this chapter. These protocols are appealing as they eliminate most of the space overhead when compared to directory protocols. However, they suffer from the problem of being bandwidth intensive because of requiring broadcast messages from the home tile when coherence maintenance is necessary. To alleviate such an issue, Lodde et al. [96] advocate the use of an heterogeneous NoC design that is composed of the conventional interconnection network coupled with a dedicated gather control network. The latter network is used to collect all the acknowledgement messages that *Hammer* protocol produces. In this way, a significant reduction in network traffic is achieved that also entails reductions in execution time.

While our proposal also relies on a dedicated on-chip network, *ECONO* is more efficient since our special network is based on *G-Lines* technology for extremely fast broadcasts. Moreover, our proposal is simpler for the use of atomicity that guarantees correctness without requiring acknowledgement messages thus reducing network traffic.

After discussing the main categories of coherence protocols and comparing them against our *ECONO* protocol, in the following part of this section we describe a set of efficient state-of-the-art coherence protocols that show novel ideas, mechanisms, efficient infrastructures or alternative technologies, that could be applied in *ECONO* due to their orthogonal nature.

Recent advances in nanophotonic device manufacturing have developed optical technology, allowing for high-bandwidth low-latency energy-efficient global links. This technology has encouraged a return to broadcast-based coherence protocols such as *ATAC* [44] or [154]. More specifically, *ATAC* operates in the same way as a conventional limited directory, but when the capacity of the sharer list is exceeded, it resorts to very efficient optical-based broadcasts to invalidate all possible block sharers. While *ECONO* also relies on very efficient broadcasts, albeit using *G-Lines* technology, we remove all storage overhead of a directory as well as all broadcast and acknowledgement messages from the main interconnect. Moreover, Xu et al. [154] take benefit from the high bandwidth density of optical technology to integrate in their coherence protocol an optical ring in order to realize global broadcasts efficiently for a large-scale clusterized system.

Differently, *ECONO* does not require high bandwidth to broadcast messages and rather than broadcasting upon every cache miss, we do so when coherence activity is required, thereby saving power. Another significant advance in the use of nanophotonic technology is simplifying protocol design and verification to the point of approximating to the concept of the maintenance of coherence in bus-based machines. An example in this category could be *Atomic Coherence* [37]. In *Atomic Coherence*, all coherence requests perform atomically and hence, all possible racing requests are removed. To do so, a very efficient optical-based mutex is implemented. In contrast, *ECONO* enables races from the requesting cores to home tiles, and serialization is only employed to atomically broadcast express coherence notifications, thus increasing concurrency. Moreover, we do not inject an important amount of coherence-related traffic into the CMP's interconnect, hence saving on-chip traffic and energy.

Our *ECONO* protocol could be also integrated using nanophotonic technology that we could explore to provide very efficient broadcasts for the *ACN* messages, as well as concurrent broadcasting for different memory blocks without resorting to replication of resources (e.g. the two *GecNetworks* implemented).

Conscious about the ever-increasing complexity of hardware-based coherence maintenance, other proposals advocate to transfer some cache coherence management to software, mainly to the OS. The key idea behind these proposals is the flexibility achieved by software to modify the protocol, or even to fix bugs, without costlier hardware changes in exchange of a reduced performance in most cases. For instance, Fensch and Cintra [22] rely on minimal hardware extensions and the OS's virtual memory system, to map data to tiles at the granularity of pages under OS control and to support remote cache accesses in hardware. By doing so, all L1 caches are treated as a single logical cache and the proposed mechanism avoids duplicate copies of a single cache line since every memory line can only reside in one L1 cache (the home cache), and processors in other tiles must perform remote cache reads and writes to access the data. Other proposals leverage OS capabilities to improve performance of coherence protocols. Cuesta et al. [13] obtain remarkably effectiveness of directory-based protocols, particularly directory caches, by avoiding the tracking of private blocks because they do not need coherence. For that, the OS dynamically detects shared blocks at page granularity by taking advantage of existing hardware structures such as Translation Lookaside Buffers (TLBs), page tables, and Miss Status Holding Registers (MSHRs). Moreover, Subspace Snooping [33] enhances efficiency of snooping-based protocols by leveraging page tables to keep track of sharers of a

memory block at page granularity, so that snoop requests are issued only to the required cores, thereby improving network traffic, power and performance.

Our *ECONO* proposal also makes use of OS support in order to determine the sharing status of memory blocks (i.e private or shared blocks), similar to the detection mechanism proposed by [13]. In this way, coherence activity is performed just for shared blocks.

There are other protocols that leverage the possibilities that bring the interconnection networks used to convey the coherence-related traffic. For instance, Cheng et al. [89] advocate heterogeneous on-chip networks, through partitioning available metal area across different wire implementations featuring distinct latency, bandwidth and energy properties. So, every type of coherence-related traffic is mapped onto the most suited type of wire thereby yielding improved performance and energy. Moreover, Chaves et al. [144] explore to harness the physical services provided by NoC, such as multicast and priorities, to optimize a directory-based coherence protocol. In exploiting the multicast capability, a single multicast message is injected into the NoC to invalidate all block sharers in L1 caches, what saves traffic and consequently energy consumption. The use of priorities implemented in the NoC can be exploited to increase memory throughput by injecting long and short messages into high and low priority channels, respectively.

Our proposal could take advantage of this kind of optimizations to enhance performance by transmitting the coherence-related traffic, other than the *ACN* messages, through different wires. Moreover, we could exploit multicast capabilities in order to reduce the effect on power dissipation derived from the broadcast-based transmissions for the *ACN* messages.

Alternative schemes adopt different strategies for the maintenance of coherence by dynamically selecting the most appropriate coherence protocol based on application behavior. For example, ARCc [112] seamlessly combines directory and shared-NUCA based coherence protocols that co-exist in hardware. The former protocol is always the preferred choice and after a sampling period of time, in which an in-hardware analytical model monitors application characteristics, it may decide a change to the latter protocol. Moreover, Chtioui et al. [45] present an hybrid update/invalidate coherence protocol that dynamically adapts its functioning mode according to the application needs. This protocol analyzes the kind of operations sent to memory (i.e. read or write) as well as the intensity of such operations to estimate a suitable threshold to use that determines the best-performing protocol at that moment.

ECONO could also co-exist with another protocol like, for example, a directory-based protocol, which we dynamically could choose if the cost of broadcasting an *ACN* message is higher than transmitting a few point-to-point messages over the main interconnect. This could be the case of memory blocks with a few sharers.

To conclude this section, we expose a final group of interesting state-of-the-art solutions for coherence maintenance. This may suggest new solutions to be explored that may lead to even higher performance improvements for our coherence protocol.

For instance, Barrow-Williams et al. [108] present Proximity Coherence, a directory-based coherence protocol that exploits the physical locality of shared data before sending a cache miss request to the directory, hence reducing L1 cache miss latencies and network traffic. For that, the private caches of neighboring cores are probed upon the cache miss, and only if all their caches do not have a copy of the data the request is sent towards the directory. To implement this proposal, the authors employ new lightweight dedicated links to interconnect a core and its neighbors, and a graph structure embedded into every private cache to avoid inconsistencies with the directory information.

Moreover, Ros et al. [11] present direct coherence, a cache coherence protocol that reduces the number of hops in directory-based protocols to solve cache misses, thereby achieving shorter cache miss latencies. Specifically, by using special hardware structures with manageable on-chip area, requests are directed to the owner cache that provides the block in a cache miss, thus removing indirection to access the directory. To keep up-to-date coherence information, the authors also propose several policies, based on hint messages and signatures, that lead to significant performance improvements in terms of execution time and network traffic.

Besides, Huang et al. [157] identify the problem of the uneven distribution of blocks that are actually cached and tend to produce severe conflicts in sparse directories on a few homes (hot-homes). To solve that, the authors propose to extend the capacity of hot-homes by using an alternative home where state and locations of a block can be recorded. As a result, this approach achieves a significant reduction in block invalidations and in cache misses per instruction.

Finally, SWEL [129] is a coherence protocol aimed at reducing the number of coherence operations by placing data in their optimal location. For that, the protocol classifies blocks into two categories: private or read-only blocks, and shared-written blocks. As the former group does not require coherence and represent the majority of memory references, the corresponding blocks are placed at the L1 caches. The latter group are forced to reside at the shared L2 level, what

removes indirection in directory-based protocols for many common write-sharing patters like the producer-consumer pattern. In this way, the protocol is more about classifying blocks than tracking their sharing information, thereby leading to a simpler and storage-efficient protocol.

5.7 Conclusions

In this chapter we propose *ECONO*, a simple and efficient coherence protocol for many-core CMPs. To keep coherence, *ECONO* relies on express coherence notifications (*ACN* messages) which are broadcast atomically over a dedicated lightweight on-chip network leveraging *G-Lines* technology for superior efficiency.

We study a baseline implementation for *ECONO* and propose two different extensions. First, *4-hop ECONO* with imprecise coherence information to shorten the size of *ACN* messages, hence operation latency. And second, we also consider a *3-hop* optimization to reduce the number of hops and save network traffic.

Through detailed execution-driven simulations of a 16-tile CMP, we determine that the best option to implement *ECONO* consists of two 3-*G-Line*-based networks. Moreover, when considering imprecise coherence information, `Index+5` is the preferred choice. To quantify the benefits of our proposal, we compare the performance achieved with *ECONO* against two contemporary coherence protocols, *Hammer* and *Directory*. This study reveals that *ECONO* features the simplest design, requires an on-chip area overhead similar to *Hammer*, reports similar performance to *Directory*, and constitutes the most energy efficient design.

Due to the novelty of our proposal, from the discussion provided in the last section (related work), we also conclude that our *ECONO* protocol could take advantage of a number of optimizations incorporated in other coherence protocols, so that we could even enhance the promising performance results obtained in our experiments, that would be of paramount importance when considering larger many-core CMPs.

CHAPTER 6

Conclusions and Future Ways

6.1 Conclusions

Until the last decade the driving force to improve performance for each new generation of computing systems was the virtuous cycle of minimizing transistors and increasing operation frequency. As a result, uniprocessors machines became more and more complex to extract the maximum instruction-level parallelism as possible by relying on wide-issue and deep pipelines, highly-speculative and out-of-order execution, or elaborate branch predictors, reaching to the point of diminishing returns. In consequence, new architectural designs were proposed to overcome such diminishing returns and the best option was to integrate a multiprocessor in a single chip, constituting the multicore architectures (chip-multiprocessors or CMPs). CMPs were conceived to exploit thread-level parallelism (TLP) rather than instruction-level parallelism (ILP) by incorporating simpler and lower frequency cores than those complex uniprocessors systems, yielding superior efficiency considering a similar floorplan.

More and more cores are being added to these throughput-oriented machines on a single chip resulting in systems knowns as many-core CMPs. To increase scalability and to obtain a manageable complexity, many-core CMPs are built upon a tiled composition, where basic building blocks called tiles are comprised of a processing core, private levels of caches with a slice of a global shared cache, and also routing logic to interconnect all tiles to a global point-to-point network, generally following a 2D-mesh topology. To reduce programming efforts, an intuitive shared-memory programming model is commonly used where commu-

nication and synchronization operations among threads are performed by means of conventional memory access instructions using a global address space over a shared memory. Nevertheless, since 2004, there has been identified a slowdown in the exponential growth of performance because of the thermal-power constraints that preclude computer architects from relying on the virtuous cycle successfully applied in uniprocessor systems as aforementioned. To continue with an incessant performance growth, new architectural innovations are being devised and the integration of non-digital technologies along with CMOS transistors is being considered in the latest roadmap of technology, what is typically known as the *More-than-Moore* trend.

In this thesis, we have identified three fundamental performance bottlenecks in the context of many-core CMPs. Highly contended synchronization in barriers and locks constitute a big challenge to manage as the number of cores increases in many-core CMPs. Moreover, traditional implementations for the cache coherence protocol, which guarantees coherence across all levels of a memory hierarchy, have also to be reconsidered to keep up with efficiency for the communication and synchronization operations amongst threads based on the use of shared variables. We have proposed three distinct and complementary hardware-based solutions to overcome such performance bottlenecks. Moreover, we have also considered the use of non-digital technology to help us break such limitations obtaining superior efficiency and scalability. To do so, we have leveraged the full-custom state-of-the-art *G-Lines* technology, although we have also explored the efficiency of our proposals using a current standard cell design methodology. Next, we expose the main conclusions derived from our three proposals in this thesis: *GBarrier*, *GLock* and *ECONO*.

The first of our proposals (namely *GBarrier*) is aimed to overcome performance limitations of barrier operations in many-core CMPs. *GBarrier* is a novel hardware-based barrier mechanism specifically designed to enable efficient barriers by removing all performance limitations of software-based barrier implementations, and even in all hardware-barrier mechanisms to date. In particular, our *GBarrier* mechanism consists of two main components: First, a very lightweight dedicated on-chip network that could be deployed in a hierarchical layout for scalability. The second is a simple and very fast synchronization protocol implemented atop the previous infrastructure. The reason why our proposal is much more efficient is that differently to software approaches based on the use of atomic read-modify-write instructions operating on shared-memory positions, *GBarrier* does not have any influence on the memory system, hence saving traffic and energy. More specifically, we have avoided all coherence activity, barrier-related

network traffic and the involved energy consumption, that software approaches introduce and that restrict scalability. We have also proposed how to extend *GBarrier* to be easily adapted in other scenarios and system configurations: several *GBarriers*, barriers among group of cores, larger many-core CMPs and SMT processor cores. To evaluate *GBarrier*, we have considered two implementations of our infrastructure by leveraging *G-Lines* and *Standard* technologies. Our study in terms of raw performance statistics reveals that differences in on-chip area overhead and power dissipation can be considered negligible between both technologies, although, as expected, the former technology reports the minimum synchronization latency, whereas the latter leads to a cost-effective implementation. We integrate both *GBarrier* implementations into a detailed execution-driven simulator (Sim-PowerCMP) of a 32-core CMP running a set of benchmarks: kernels and scientific applications. From this study, both *GBarrier* implementations report very similar reductions in execution time, thus not making our proposal so dependent on a full-custom technology to achieve extremely efficient synchronization in many-core CMPs. In particular, for the kernels and the scientific applications under study our proposal brings average reductions of 54% and 21%, respectively, in total execution time, resulting in improved scalability for the applications. We also have obtained reductions of 53% and 18%, respectively, in network traffic. The reason is that our proposal does not rely on shared memory positions and the cache coherence protocol saves a significant amount of messages on the main interconnection network. Finally, all these gains lead to improvements of 76% and 31%, respectively, in the energy-delay^2 product (ED^{2}P) metric for the full CMP.

Regarding lock synchronization, we have identified that contention is a key constraint to performance and scalability when there are a significant amount of threads willing to access into the same CS at once. To achieve a fair, very efficient and scalable solution for locks that are highly contended, we have proposed *GLock*. *GLock* is based on a dedicated on-chip network and relies on a simple token-based messaging-protocol. Due to the fact that the actual problem of this kind of synchronization mechanism occurs for high contention, our proposal could be combined with a software-based implementation for low contention (e.g. *Simple Locks* enhanced with the `test-and-test&set` optimization). Moreover, a deep analysis of some relevant benchmarks discloses a reduced number of highly-contended locks in most cases, so that replication of the *GLock*'s resources is not expected to be a constraint. As for *GBarrier*, to evaluate *GLock*, we have made use of *G-Lines* and *Standard* technology coming to the very same conclusions in terms of raw performance: negligible on-chip area overhead and power

dissipation in both technologies, cost-effective implementation for *Standard*, and faster synchronization latency for the *G-Lines* technology. We integrate both *GLock* implementations into Sim-PowerCMP, and discuss synchronization efficiency results as compared to the most efficient software-based lock implementation. To do so, we have simulated a 32-core CMP with a 2D-mesh data network and employ a set of microbenchmarks and real applications. Both *GLock* implementations report very similar reductions in execution time, hence not making our proposal so dependent on a full-custom technology. From our evaluation, the next significant average reductions for the microbenchmarks and the real applications respectively are achieved: 42% and 14% in execution time; 76% and 23% in network traffic; and 78% and 28% in the ED^2P metric for the full CMP.

Finally, we have also proposed a novel implementation for a coherence protocol in the context of many-core CMPs, namely *ECONO*. Our proposal makes use of express coherence notifications (*ACN* messages) which are atomically broadcast over a dedicated lightweight on-chip network. We have only considered the *G-Lines* technology since the *Standard* one has severe technical constraints to provide the required scalability and lightweight implementation to build a broadcast-based network on chip. *ECONO* has been studied starting with a baseline implementation, called *4-hop ECONO*. This implementation requires four hops in the critical path to resolve an L1 cache write miss that a processing core gets and several sharers of the block exist so that their copies must be invalidated. In addition, we have proposed two different extensions of this first version for superior efficiency. First, imprecise coherence information to shorten the size of *ACN* messages and operation latency of a *4-hop ECONO* implementation. And second, we have reduced the number of hops in the critical path of *ECONO* to implement a *3-hop* optimization, thereby leading to savings in network traffic and energy. To evaluate *ECONO* we have employed the full-system simulation tool Simics-GEMS, and we have considered a 16-tile CMP. From our experimental study in terms of raw performance statistics, we have concluded that the best option to implement *ECONO* consists of two 3-*G-Line*-based network, and when considering imprecise coherence information, `Index+5` is the preferred choice. We have quantified performance benefits from our proposal by comparing performance results against two contemporary coherence protocols, *Hammer* and *Directory*. The main outcomes of this study are the following: *ECONO* features the simplest design, requires an on-chip area overhead similar to *Hammer*, reports similar performance to *Directory*, and constitutes the most energy efficient design. Due to the simplicity of our proposal, we have identified that diverse orthogonal optimizations could be easily applied that are present in contemporary protocols

such as filtering, to minimize the impact on energy of the atomic broadcast transmissions, or avoiding the indirection to home tiles by adding extra information into the *ACN* messages (the next block owner), so that all the L1$ sharers always know who is the next owner for the memory block.

As a final conclusion from all above, we can affirm that our proposals represent a step forward towards the resolution of the challenges that many-core CMP architectures will pose to computer architects.

6.2 Future Ways

The results presented in this thesis open a number of new research paths to explore. Amongst them, we identify the following.

We could investigate the natural extension of the work carried out in this thesis by the combination of all of the proposals presented here to improve the efficiency of many-core CMPs. In this way, synchronization operations for barriers and highly-contended locks would be implemented by using *GBarrier* and *GLock*, whereas the coherence protocol to be used would be *ECONO*. Notice that, the contention of the *ECONO*'s network to transmit the *ACN* messages would be reduced, since the maintenance of coherence for all shared variables affected by the software-based barriers and locks considered in our evaluation would be removed. It could be also very interesting quantifying the magnitude of these savings and the resulting performance improvements.

Moreover, while our hardware-based synchronization mechanisms successfully work for a 32-core CMP architecture, an appealing question to answer as future work could be: *What is the maximum scalability that the current versions of GBarrier and GLock could reach?* In this way, we would determine the limits of our proposals and whether it is possible to keep up with scalability in a future 1000-core CMP platform, like many other recent proposals are already attempting to envision (e.g. in the context of coherence protocols [44, 61]). For that, as proposed in the corresponding chapters of this thesis that explain how to deal with larger many-core CMPs, we could adopt a clusterization strategy and/or build hierarchical layouts.

Another important question to answer could be: *What is the limit of replication for GBarrier and GLock?* For the representative set of benchmarks that we have employed to evaluate our proposals and for the simulated 32-core CMP, we have been required to replicate resources only once to achieve the maximum efficiency. The reason is that our proposals have been specifically devised to operate in high

contention scenarios either to perform a barrier, or to take exclusive access into a critical section protected by a lock, where all threads/cores are involved in the operation. An interesting research path could be evaluating performance when considering subsets of the total amount of CMP's cores. Here, replication of resources is strictly necessary to make possible the concurrency of barriers or locks in the same or among different applications.

Other important ways to explore in the future are those related to extend the applicability of *GBarrier* and *GLock* in other contexts or scenarios:

- *Other Programming Models*. To design and evaluate our proposals, we rely on a shared-memory programming model and the benchmarks are written using Pthreads. This requires a hand-made process to insert the corresponding function calls in the benchmark's code to the library that encapsulates the functionality that communicates with our hardware infrastructures. In the same way that we presented in [71] for *GBarrier*, we could also integrate our *GLock* mechanism into the widespread OpenMP. By doing so, parallelism is specified at a very high level by inserting directives (pragmas) to a sequential `C` program, and a compiler is responsible for translating these directives into parallel threads of execution. Then, our synchronization functionality would be implemented within the OpenMP's runtime library, that would be queried by the parallel threads. In this way, programmers could take benefit from our proposals directly without any further programming effort. Moreover, we could also evaluate how to deal with synchronization operations in nested parallelism for a clusterized system. In nested parallelism, a first level of parallelism is used to distribute coarse-grained tasks to clusters, and these tasks are comprised of one or more inner levels of fine-grained (e.g. loop-level) parallelism that is distributed to cores within a cluster [7]. In particular, fine-and-coarse-grained barrier synchronization would be implemented by using different *GBarriers*. Besides, the distribution of the different tasks could be implemented by using a *GLock*, where different computing units (cores) could take exclusive access into the pool of tasks to fetch a new task to process.

  Furthermore, we could integrate our proposals in the context of the StarSs programming model [137]. Here, the programmer use OpenMP-like pragmas to define tasks and their inputs and outputs. A source-to-source translator and a runtime system are responsible for scheduling the tasks to be executed preserving dependencies among them. There are multiple instantiations for StarSs that include GRIDSs [153] (for the Grid), CellSs [113] (for

the Cell B.E. [62]), SMPSs [134] (for multicore processors) and GPUSs [38] (for GP-GPUs). Since there could be different groups of tasks that need to be synchronized, we could extend this task-based programming model to make use of our *GBarrier*. Besides, for the pool of tasks submitted to every particular computing unit, similar to commented above, the process of fetching a new task could be implemented by using a *GLock*. In this new scenario, the CS to be protected would be the pool of tasks, and the different cores would fetch a new task in mutual exclusion through our *GLock*.

Another programming model in which we could integrate our synchronization mechanisms would be a message-passing programming model [29]. In particular, the standardized and portable implementation called *Message Passing Interface* [102] (MPI). In this programming model, each process executes in its own address space and communication and synchronization operations among processes are based on explicit *send* and *receive* messages. In its simplest from, *send* specifies a local data buffer that is to be transmitted and a receiving process, and *receive* specifies a sending process and a local data buffer into which the transmitted data is to be placed. Then, the matching *send* and *receive* causes a data transfer from one process to another. Notice that, since our synchronization mechanisms also operate on a message-passing protocol, we could easily integrate them into MPI. For example, *GBarrier* could be used to enhance performance of the MPI's barrier primitive called `MPI_Barrier`, or *GLock* could be employed to improve locks in a threaded MPI library [105]. Additionally, we could also consider to improve some of the algorithms used in the MPI's communication operations to send collective and individual messages among processes. For example, the `MPI_Bcast` function, that implicitly involves a barrier operation to determine whether all the destination buffers at all the receiver processes are available to start with the broadcast communication, could be improved by using our *GBarrier* proposal. Moreover, since our on-chip infrastructures are composed of point-to-point links, we could map the MPI's point-to-point communication primitives onto them, for example to improve the blocking `MPI_Send` function.

- *Other Platforms*. In this thesis, we have focused on tiled many-core CMPs but we could have considered other type of architectures such as GP-GPU (Tesla [111]), asymmetric or heterogeneous processors (Cell B.E.) and clusterized systems (Platform 2012 [135]), amongst others. To do so, the imple-

mentation of the controllers and the library-level code required by *GBarrier* and *GLock* presented in this thesis have to be properly adapted to the target platform.

- *Other Technologies*. With the advent of the *More-than-Moore* era [150], chip-makers will incorporate non-digital technology in next-generation many-core CMPs. In this thesis, we have also utilized non-digital technology (*G-Lines*) to implement all our proposals. As future work, we could also take into account the benefits provided by emerging nanophotonic technology [123] to implement our proposals. The high bandwidth density and very low latency that feature this technology would preclude our mechanisms from replication of resources when it is necessary to share the use of them at the same time. Limits of scalability could be other interesting study to be done using this technology.

- *Other Purposes*. We could also consider to extend the applicability of *GLock* in the context of hardware transactional memory. In particular, we would provide a hardware-based implementation of the *precommit* phase of a *Lazy-Lazy* system, by efficiently and fairly selecting just one of the transactions aimed to commit using an infrastructure similar to the *GLock*, so that this transaction could safely proceed to make their changes visible.

In regards to the coherence protocol we have proposed in this thesis, *ECONO*, there are also a number of future directions to investigate. To minimize impact on energy consumption for the atomic broadcast messages transmitted to ensure coherence (*ACN* messages), we could apply different types of filters other than those that have been considered. For instance, *destination filters*, like that employed in snooping protocols [147], could save cache-tag look-up power dissipation at the L1 caches when the corresponding L1 controllers receive the *ACN* message and the requested block is not present in their L1 cache. Moreover, in-network filters [106] could also be applied in order to enable unicast or multicast transmissions in the *ECONO*'s network. This considerably would save energy in case of maintaining coherence for a reduced number of block's sharers.

Other extensions for *ECONO* could be the following:

- *Clusterization of ECONO*. We could employ a hierarchical strategy to scale the infrastructure of *ECONO* and then, there could be disjoint coherence domains at both inter-and intra-cluster levels to manage, running different applications (multiprogrammed workloads) or different virtual machines for server consolidation [101].

- *Direct-ECONO*. We also propose to extend *ECONO* by removing indirection to home nodes that exhibit *Directory* protocols. In this way, like [11], cores that suffer an L1 cache miss would send a point-to-point request towards the owner cache, rather than sending the corresponding message to the home tile. For that, when a core gets a write cache miss and transmits the corresponding message to the home tile, the latter would add to the broadcast *ACN* message the core's ID. Then, all the remaining L1 caches would know who would become the next block owner, and future cache misses would be directed to the owner.

- *Other Technologies*. As for *GBarrier* and *GLock*, we could also explore the implementation of *ECONO* using alternative technologies such as nanophotonic technology. In addition, we could determine maximum scalability for the resulting infrastructure.

- *Hybrid Coherence*. We could make use of an hybrid coherence protocol [45] that would employ different protocols depending for example on the degree of data sharing present in the parallel application. A runtime would be implemented in order to dynamically choose the best protocol in each case. For example, by combining *Directory* and *ECONO*, the former protocol would be the preferred choice when there are a few sharers, thus avoiding the costlier *ACN* broadcast.

Bibliography

[1] A. Agarwal, R. Simoni, J. Hennessy and M. Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, 1988. 5.2.1, 5.2.2, 5.3

[2] A. Ahmed, P. Conway, B. Hughes and F. Weber. AMD Opteron Shared Memory MP Systems. In *Proceedings of the HotChips Symposium*, 2002. 5.1, 5.2.1, 5.5.1

[3] A. Flores, J. L. Aragón and M. E. Acacio. Sim-PowerCMP: A Detailed Simulator for Energy Consumption Analysis in Future Embedded CMP Architectures. In *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops*, 2007. 2.2.1, 3.4.1, 4.1

[4] A. Gupta, W. Weber and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Schemes. In *Proceedings of the 8th International Conference on Parallel Processing*, 1990. 5.6

[5] A. Hesseldahl. Moore's Law Is Alive and Well, and Intel Will Prove It Today, 2011. http://allthingsd.com/20110504/moores-law-is-alive-and-well-and-intel-will-prove-it-today. 1.1

[6] A. Kägi, D. Burger and J. R. Goodman. Efficient Synchronization: Let Them Eat QOLB. In *Proceedings of the 24th International on Computer Architecture*, 1997. 4.5

[7] A. Marongiu, P. Burgio and L. Benini. Fast and Lightweight Support for Nested Parallelism on Cluster-Based Embedded Many-Cores. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, 2012. 6.2

[8] A. Marowka. Back to Thin-Core Massively Parallel Processors. *Computer*, 99, 2011. 2.2.2

[9] A. P. Jose and K. L. Shepard. Distributed Loss-Compensation Techniques for Energy-Efficient Low-Latency On-Chip Communications. *IEEE Journal of Solid State Circuits*, 42(6):1415–1424, 2007. 1.6

[10] A. R. Alameldeen and D. A. Wood. IPC Considered Harmful for Multiprocessor Workloads. *IEEE Micro*, 26(4):8–17, 2006. 2.4

[11] A. Ros, M. E. Acacio and J. M. García. A Direct Coherence Protocol for Many-Core Chip Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1779–1792, 2010. 5.6, 6.2

[12] B. Beck, B. Kasten and S. Thakkar. VLSI Assist for a Multiprocessor. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating System*, 1987. 3.5

[13] B. Cuesta, A. Ros, M. E. Gómez, A. Robles and J. Duato. Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks. In *Proceedings of the 37th International Symposium on Computer Architecture*, 2011. 5.3.2, 5.6

[14] B-H. Lim and A. Agarwal. Reactive Synchronization Algorithms for Multiprocessors. *ACM SIGPLAN Notices*, 29(11):25–35, 1994. 4.5

[15] B. M. Beckmann and D. A. Wood. TLC: Transmission Line Caches. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011. 3.5

[16] B. Rooney. ARM CEO: Moore's law "alive and well", 2011. http://blogs.wsj.com/tech-europe/2011/03/15/arm-ceo-moores-law-alive-and-well. 1.1

[17] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962. 2.3.3

[18] C. Bienia, S. Kumar, J. P. Singh and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the IEEE International Conference on Parallel Architectures and Compilation Techniques*, 2008. 2.3.3, 5.5.1

[19] C-C. Kuo, J. B. Carter and R. Kuramkote. MP-LOCKs: Replacing H/W Synchronization Primitives with Message Passing. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, 1999. 4.5

[20] C. Cascaval, J. G. Castaños, L. Ceze, M. Denneau, M. Gupta, D. Lieber, J. E. Moreira, K. Strauss and H. S. Warren. Evaluation of a Multithreaded Architecture for Cellular Computing. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, 2002. 3.5

[21] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S. W. Yang and R. Zak. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, 1992. 3.5

[22] C. Fensch and Marcelo Cintra. An OS-Based Alternative to Full Hardware Coherence on Tiled CMPs. In *Symposium on High-Performance Computer Architecture*, 2008. 5.6

[23] C. J. Beckmann and C. D. Polychronopoulos. Fast Barrier Synchronization Hardware. In *Proceedings of the 2nd Conference on Supercomputing*, 1990. 3.5

[24] C. J. Hughes, V. S. Pai, P. Ranganathan and S. V. Adve. RSIM: Simulating Shared-Memory Multiprocessors with ILP Processors. *IEEE Computer*, 35(2):40–49, 2002. 2.2.1

[25] C. Wagner and F. Mueller. Token-based Read/Write-Locks for Distributed Mutual Exclusion. In *Proceedings of the 6th International Euro-Par Conference on Parallel Processing*, 2000. 4.5

[26] Cadence. SoC Encounter. http://www.cadence.com/products /di/-soc_encounter/pages/default.aspx. 2.2.3

[27] D. Brooks, V. Tiwari and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture*, 2000. 2.2.1

[28] D. E. Culler, A. C. Aparci-Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. V. Eicken and K. A. Yelick. Parallel Programming in Split-C. In *Proceedings of the ACM/IEEE International Conference on SuperComputing*, 1993. 2.3.3

[29] D. E. Culler, J. P. Singh and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1998. (document), 1, 1.2, 2.2.1, 3.1, 3.4.1, 4.2.2, 4.5, 6.2

[30] D. Geer. Industry Trends: Chip Makers Turn to Multicore Processors. *Computer*, 38:11–13, 2005. 1.1

[31] D. J. Sorin, M. D. Hill and D. A. Wood . *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture, 2011. 5.1

[32] D. J. Sorin, M. D. Hill and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture#16, 2011. 5.3.2

[33] D. Kim, J. Ahn, J. Kim and J. Huh. Subspace Snooping: Filtering Snoops with Operating System Support. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, 2010. 5.6

[34] D. Ludovici, G. N. Gaydadjiev, D. Bertozzi and L. Benini. Capturing Topology-Level Implications of Link Synthesis Techniques for Nanoscale Networks-on-Chip. In *Proceedings of the 19th ACM Great Lakes Symposium on VLSI*, 2009. 3.2.6.2

[35] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling Ways and Associativity. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010. 5.6

[36] D. Sanchez and C. Kozyrakis. SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*, 2012. 5.6

[37] D. Vantrease, M. H. Lipasti and N. Binkert. Atomic Coherence: Leveraging Nanophotonics to Build Race-Free Cache Coherence Protocols. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*, 2011. 5.1, 5.3, 5.6

[38] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo and E.S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Proceedings of the 6th International Euro-Par Conference on Parallel Processing*, 2009. 6.2

[39] E. Mensink, D. Schinkel, E. Klumperink, E. Tuijl and B. Nauta. A 0.28pf/b 2gb/s/ch Transceiver in 90nm CMOS for 10mm On-Chip Interconnects. In *Proceedings of the IEEE Solid-State Circuits Conference*, 2007. 1.6

[40] E. W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, 8(9):569, 1965. 4.1

[41] F. H. McMahon. Livermore Fortran Kernels: A Computer Test of Numerical Performance Range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, 1986. http://www.netlib.org/benchmark/livermorec. 2.3.2, 3.4.1

[42] G. E. Moore. Cramming more Components onto Integrated Circuits. *Electronics*, 38(8):114–117, 1965. (document), 1.1

[43] G. Hinton, D. Sager, M. Upton, D. Boggs, Desktop Platforms Group and Intel Corp. The Microarchitecture of the Pentium-4 Processor. *Intel Technology Journal*, 1:1–13, 2001. 1.1

[44] G. Kurian, J. E. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. C. Kimerling and A. Agarwal. ATAC: A 1000-Core Cache-Coherent Processor with On-Chip Optical Network. In *Proceedings of the IEEE International Conference on Parallel Architectures and Compilation Techniques*, 2010. 5.3.3, 5.6, 6.2

[45] H. Chtioui, R. B. Atitallah, S. Niar, J-L. Dekeyser and M. Abid. A Dynamic Hybrid Cache Coherency Protocol for Shared-Memory MPSoC. In *Proceedings of the 12th EUROMICRO Conference on Digital System Design/ Architectures, Methods and Tools*, 2009. 5.6, 6.2

[46] H. Dwyer and H. C. Torng. An Out-of-Order Superscalar Processor with Speculative Execution and Fast, Precise Interrupts. In *Proceedings of the 25th IEEE/ACM International Symposium on Microarchitecture*, 1992. 1.1

[47] H. Franke, R. Russell and M. Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Proceedings of the Ottawa Linux Symposium*, 2002. 4.2.3

[48] H. Ito, M. Kimura, K. Miyashita, T. Ishii, K. Okada and K. Masu. A Bidirectional-and Multi-Drop-Transmission-Line Interconnect for Multipoint-to-Multipoint On-Chip Communications. *IEEE Journal of Solid State Circuits*, 43(4):1020–1029, 2008. 1.6

[49] H. S. Wang, X. Zhu, L. S. Peh, and S. Malik. Orion: A Power-Performance Simulator for Interconnection Networks. In *Proceedings of the 35th IEEE/ACM International Symposium on Microarchitecture*, 2002. 2.2.1

[50] H. T. Olnowich. ALLNODE Barrier Synchronization Network. In *Proceedings of the 9th International Parallel Processing Symposium*, 1995. 3.5

[51] H. Zhao, A. Shriraman, S. Dwarkadas and V. Srinivasan. SPATL: Honey, I Shrunk the Coherence Directory. In *Proceedings of the IEEE International Conference on Parallel Architectures and Compilation Techniques*, 2011. 5.6

[52] HP Labs. CACTI, 2012. http://www.hpl.hp.com/research/cacti/. 2.2.1, 3.3.2, 4.3.2, 5.4.3

[53] Intel. Intel Unveils New Product Plans for High-Performance Computing, 2010. http://www.intel.com/ pressroom/archive/releases/2010/20100531comp.htm. 2.1

[54] Intel. The SCC Platform Overview, 2010. http://techresearch.intel.com/ spaw2/uploads/files/ SCC_Platform_Overview.pdf. 2.1

[55] Intel Labs. Single-chip Cloud Computer, 2009. http://techresearch.intel.com/ articles/Tera-Scale/1826.htm. 1.1

[56] International Technology Roadmap for Semiconductors. http://www.itrs.net/Links/2011ITRS/Home2011.htm. 1.1, 1.6

[57] J. B. Carter, C-C. Kuo and R. Kuramkote. A Comparison of Software and Hardware Synchronization Mechanisms for Distributed Shared Memory Multiprocessors. Technical Report UUCS-96-011, University of Utah Computer Science Department, 1996. 4.5

[58] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson and R. Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart & Winston, 1987. 2.2.1

[59] J. Eastep, D. Wingate, M. D. Santambrogio and A. Agarwal. Smartlocks: Self-Aware Synchronization through Lock Acquisition Scheduling. In *Proceedings of the 7th IEEE/ACM International Conference on Autonomic Computing and Communications*, 2009. 4.5

[60] J. F. Cantin, M. H. Lipasti and J. E. Smith. Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. In *Proceedings of the 32nd International Symposium on Computer Architecture*, 2005. 5.6

[61] J. H. Kelm, M. R. Johnson, S. S. Lumetta and S. J. Patel. WAYPOINT: Scaling Coherence to 1000-core Architectures. In *Proceedings of the IEEE International Conference on Parallel Architectures and Compilation Techniques*, 2010. 6.2

[62] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, 2005. 6.2

[63] J. L. Abellán, J. Fernández and M. E. Acacio. A G-Line-based Network for Fast and Efficient Barrier Synchronization in Many-Core CMPs. In *Proceedings of the 4th Workshop on Interconnection Network Architectures: On-Chip, Multi-Chip*, 2010. 1.7

[64] J. L. Abellán, J. Fernández and M. E. Acacio. A G-line-based Network for Fast and Efficient Barrier Synchronization in Many-Core CMPs. In *Proceedings of the 39th International Conference on Parallel Processing*, 2010. 1.7

[65] J. L. Abellán, J. Fernández and M. E. Acacio. A Novel Hardware-based Barrier Synchronization for Many-Core CMPs. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, 2010. 1.7

[66] J. L. Abellán, J. Fernández and M. E. Acacio. Efficient Hardware Support for Lock Synchronization in Many-core CMPs. In *Proceedings of the 12th Jornadas de Paralelismo*, 2011. 1.7

[67] J. L. Abellán, J. Fernández and M. E. Acacio. GLocks: Efficient Support for Highly-Contended Locks in Many-Core CMPs. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*, 2011. Best Paper Award in the Architectures Track. 1.7

[68] J. L. Abellán, J. Fernández and M. E. Acacio. Design of an Efficient Communication Infrastructure for Highly-Contended Locks in Many-Core CMPs. *Journal of Parallel and Distributed Computing*, 2012. DOI: 10.1016/j.jpdc.2012.06.010. 1.7

[69] J. L. Abellán, J. Fernández and M. E. Acacio. Efficient Hardware Barrier Synchronization in Many-Core CMPs. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1453–1466, 2012. 1.7

[70] J. L. Abellán, J. Fernández and M. E. Acacio. Infraestructuras de Barrera Eficientes para Sistemas Clusterizados MPSoC. In *Proceedings of the 13th Jornadas de Paralelismo*, 2012. 1.7

[71] J. L. Abellán, J. Fernández, M. E. Acacio, D. Bertozzi, D. Bortolotti, A. Marongiu and L. Benini. Design of a Collective Communication Infrastructure for Barrier Synchronization in Cluster-Based Nanoscale MPSoCs. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, 2012. 1.7, 3.2.6.2, 3.4.3.1, 6.2

[72] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 4th edition, 2006. 1.1, 1.2

[73] J. L. Shin, K. Tam, D. Huang, B. Petrick, H. Pham, C. Hwang, H. Li, A. Smith, T. Johnson, F. Schumacher, D. Greenhill, A. S. Leon and A. Strong. A 40nm 16-core 128-thread CMT SPARC SoC Processor. In *Proceedings of the International Solid-State Circuits Conference Digest of Technical Papers*, 2010. 1.1

[74] J. Laudon and D. Lenoski. The SGI Origin: A cc-NUMA Highly Scalable Server. In *Proceedings of the 24th International on Computer Architecture*, 1997. 1.5

[75] J. Leverich, A. Firoozshahian, H. Arakida, M. Horowitz, A. Solomatnikov and C. Kozyrakis. Comparing Memory Systems for Chip Multiprocessors. In *Proceedings of the 34th International Symposium On Computer Architecture*, 2007. 1.2

[76] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991. 2.2.1, 3.1, 4.5

[77] J. M. Mellor-Crummey and M. L. Scott. Synchronization without Contention. *ACM SIGARCH Computer Architecture News*, 19(2):269–278, 1991. (document)

[78] J. Mauro, R. McDougall. *Solaris Internals: Core Kernel Components*. Sun Microsystem Press, 2001. 4.2.3

[79] J. Oh, M. Prvulovic and A. Zajic. TLSync: Support for Multiple Fast Barriers Using On-Chip Transmission Lines. In *Proceedings of the 38th International Symposium on Computer Architecture*, 2011. 3.5

[80] J. P. Lozi, G. Thomas, J. Lawall and G. Muller. Efficient Locking for Multicore Architectures. Technical Report RR-7779, INRIA, 2011. 4.5

[81] J. R. Goodman, M. K. Vernon and P. J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 1989. 3.5

[82] J. Sampson, R. González, J. F. Collard, N. P. Jouppi, M. Schlansker and B. Calder. Exploiting Fine-Grained Data Parallelism with Chip Multiprocessors and Fast Barriers. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006. 2.3.2, 3.4.1, 3.5

[83] J. Sartori and R. Kumar. Low-Overhead, High-Speed Multi-core Barrier Synchronization. In *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers*, 2010. 3.1, 3.5

[84] J. Zebchuk, V. Srinivasan, M. K. Qureshi and A. Moshovos. A Tagless Coherence Directory. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*, 2009. 5.6

[85] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Willians and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences. University of California at Berkeley, 2006. 1.1

[86] K. D. Bosschere, W. Luk, X. Martorell, N. Navarro, M. O'Boyle, D. Pnevmatikatos, A. Ramirez, P. Sainrat, A. Seznec, P. Stenstrom and T. Olivier. High-Performance Embedded Architecture and Compilation Roadmap. In *Transactions on HiPEAC I*, 2007. 1.1

[87] K. Gharachorloo, D. Lenosky, J. Laudon, P. Gibbons, A. Gupta and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, 1990. 1.5

[88] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the IEEE International Symposium on Computer Architecture*, 2000. 5.1, 5.2.2, 5.5.1

[89] L. Cheng, N. Muralimanohar, K. Ramani, R. Balasubramonian and J. B. Carter. Interconnect-Aware Coherence Protocols for Chip Multiprocessors.

In *Proceedings of the 33rd International Symposium on Computer Architecture*, 2006. 5.6

[90] L. I. Kontothanassis and M. L. Scott. Efficient shared memory with minimal hardware support. *SIGARCH Computer Architecture News*, 23(4):29–35, 1995. 1.2

[91] L. M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, 27(12):1112–1118, 1978. 5.2.2, 5.6

[92] M. Azimi, N. Cherukuri, D. N. Jayasimha, A. Kumar, P. Kundu, S. Park, I. Schoinas and A. S. Vaidya. Integration challenges and tradeoffs for tera-scale architectures. *Intel Technology Journal*, 11(3):173–184, 2007. 1.1

[93] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, J.-W. Lee, P. Johnson, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe and A. Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. *IEEE Micro*, 22(2):25–35, 2002. 1.1

[94] M. Ferraresi, G. Gobbo, D. Ludovici and D. Bertozzi. Bringing Network-on-Chip Links to 45nm. In *Proceedings of the International Symposium on System on Chip*, 2011. 3.2.6.2

[95] M. L. Scott and W. N. Scherer. Scalable Queue-Based Spin Locks with Timeout. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2001. 4.2.3

[96] M. Lodde, J. Flich and M. E. Acacio. Heterogeneous NoC Design for Efficient Broadcast-based Coherence Protocol Support. In *Proceedings of the 6th IEEE/ACM International Symposium on Networks on Chip*, 2012. 5.6

[97] M. M. K. Martin, D. J. Sorin and M. D. Hill. Why On-Chip Cache Coherence is Here to Stay. *Communications of the ACM*, 55(7):78–89, 2012. 5.1

[98] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill and D. A. Wood. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) toolset. *Computer Architecture News*, 33(4):92–99, 2005. 2.2.2, 5.5.1

[99] M. Makhaniok and R. Männer. Hardware Synchronization of Massively Parallel Processes in Distributed Systems. In *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks*, 1997. 3.5

[100] M. Monchiero, G. Palermo, C. Silvano and O. Villa. An Efficient Synchronization Technique for Multiprocessor Systems on-Chip. *ACM SIGARCH Computer Architecture News*, 34(1):33–40, 2006. 3.5, 4.5

[101] M. R. Marty and M. D. Hill. Virtual Hierarchies to Support Server Consolidation. In *Proceedings of the 34th International Symposium on Computer Architecture*, 2007. 6.2

[102] M. Snir, J. Dongarra, J. S. Kowalik, S. Huss-Lederman, S. W. Otto and D. W. Walker. *MPI: The Complete Reference (2-volume set)*. The MIT Press, 1998. 6.2

[103] M. Zhang and K. Asanovic. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In *Proceedings of the 32nd International Symposium on Computer Architecture*, 2005. 1.1

[104] ModelSim. http://model.com. 2.2.3

[105] MPI-2 Journal of Development. www.mpi-forum.org/docs/mpi-20-jod.ps. 6.2

[106] N. Agarwal, L-S. Peh and N. K. Jha. In-Network Coherence Filtering: Snoopy Coherence without Broadcasts. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009. 5.6, 6.2

[107] N. Agarwal, T. Krishna, L-S. Peh and N. K. Jha. GARNET: A Detailed On-Chip Network Model Inside a Full-System Simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2009. 2.2.2, 5.5.1

[108] N. B. Williams, C. Fensch and S. Moore. Proximity Coherence for Chip Multiprocessors. In *Proceeding of IEEE International Conference on Parallel Architectures and Compilation Techniques*, 2010. 5.6

[109] N. P. Jouppi and S. J. E. Wilton. Tradeoffs in Two-Level On-Chip Caching. In *Proceedings of the 21st International Symposium on Computer Architecture*, 1994. 1.1

[110] N. R. Tallent, J. M. Mellor-Crummey and A. Porterfield. Analyzing Lock Contention in Multithreaded Applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010. 4.1, 4.2.3

[111] NVIDIA Tesla GPGPU series. http://www.nvidia.com/object/tesla-supercomputing-solutions.html. 6.2

[112] O. Khan, H. Hoffmann, M. Lis, F. Hijaz, A. Agarwal and S. Devadas. ARCc: A Case for an Architecturally Redundant Cache-coherence Architecture for Large Multicores. In *Proceedings of the International Conference on Computer Design*, 2011. 5.6

[113] P. Bellens, J. M. Perez, R. M. Badia and J. Labarta. CellSs: a Programming Model for the Cell BE Architecture. In *Proceedings of the International Conference on Supercomputing*, 2006. 6.2

[114] P. Bright. IBM's New Transactional Memory: Make-or-Break Time for Multithreaded Revolution, 2011. http://arstechnica.com/hardware/news/2011/08. 1.1

[115] P. Conway. Blade Computing with the AMD Magny-Cours Processor. In *Proceedings of the 21st Symposium on High Performance Chips*, 2009. 1.1

[116] P. Coteus, H. R. Bickford, T. M. Cipolla, P. G. Crumley, A. Gara, S. A. Hall, G. V. Kopcsay, A. P. Lanzetta, L. S. Mok, R. Rand, R. Swetz, T. Takken, P. La Rocca, C. Marroquin, P. R. Germann and M.J. Jeanson. Packaging the Blue Gene/L Supercomputer. *IBM Journal of Research and Development*, 49(2):213–248, 2005. 3.5

[117] P. Marcuello, A. González and J. Tubella. Speculative Multithreaded Processors. In *Proceedings of the 12th International Conference on Supercomputing*, 1998. 1.1

[118] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, 2002. 2.2.2, 5.5.1

[119] P. Tang and P. C. Yew. Processor Self-Scheduling for Multiple-Nested Parallel Loops. In *Proceedings of the the International Conference on Parallel Processing*, 1986. 3.1

[120] Plurality. The HyperCore Processor. http://www.plurality.com/hypercore.html. 3.2.6.2

[121] R. Golla. Niagara2: A Highly Threaded Server-on-a-Chip, 2006. http://www.opensparc.net/pubs/preszo/06/04-Sun-Golla.pdf. 2.1

[122] R. Ho, T. Ono, R. D. Hopkins, A. Chow, J. Schauer, F. Y. Liu and R. Drost. High-Speed and Low-Energy Capacitively-Driven On-Chip Wires. *IEEE Journal of Solid State Circuits*, 43(1):52–60, 2008. 1.6

[123] R. Kirchain and L. Kimerling. A Roadmap for Nanophotonics. *Nature Photonics*, 1(6):303–305, 2007. 3.2.6.2, 4.2.6.2, 5.3.3, 5.4.4, 6.2

[124] R. Kumar, V. Zyuban and D. M. Tullsen. Interconnections in Multi-core Architectures. In *Proceedings of the 32nd International Symposium on Computer Architecture*, 2005. 1.1

[125] R. Rajwar and J. R. Goodman. Transactional Lock-free Execution of Lock-based Programs. In *Proceedings of the 10th Annual Conference on Architectural Support for Programming Languages and Operating Systems*, 2002. 2.3.1

[126] R. T. Chang, N. Talwalkar, P. Yue and S. S. Wong. Near Speed-of-Light Signaling over On-Chip Electrical Interconnects. *IEEE Journal of Solid-State Circuits*, 38(5):834–838, 2003. 1.6

[127] S. Borkar and A. A. Chien. The Future of Microprocessors. *Communications of the ACM*, 54(5):67–77, 2011. 1.3

[128] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh and A. Gupta. The SPLASH-2 programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995. 2.3.3, 4.1, 4.4.1, 5.5.1

[129] S. H. Pugsley, J. B. Spjut, D. W. Nellans and R. Balasubramonian. SWEL: Hardware Cache Coherence Protocols to Map Shared Data onto Shared Caches. In *Proceeding of IEEE International Conference on Parallel Architectures and Compilation Techniques*, 2010. 5.6

[130] S. Kumar, D. Jiang, R. Chandra and J. P. Singh. Evaluating Synchronization on Shared Address Space Multiprocessors: Methodology and Performance. *ACM SIGMETRICS Performance Evaluation Review*, 27(1):23–34, 1999. 2.3.1

[131] S. L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996. 3.5

[132] S. Shang and K. Hwang. Distributed Hardwired Barrier Synchronization for Scalable Multiprocessor Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 6(6):591–605, 1995. 3.5

[133] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, 1996. 1.5

[134] SMPSs Programming Model. http://www.bsc.es/computer-sciences/programming-models/smp-superscalar/programming-model. 6.2

[135] ST Microelectronics and CEA. Platform 2012: A Many-core Programmable Accelerator for Ultra-Efficient Embedded Computing in Nanometer Technology, 2012. http://www.2parma.eu/images/stories/p2012_whitepaper.pdf. 3.2.6.2, 6.2

[136] Standard Performance Evaluation Corporation. SPEC CFP95 Benchmarks. http://www.spec.org/cpu95/CFP95/. 2.3.3

[137] StarSs Programming Model. http://www.ccs.tsukuba.ac.jp/ people/hac2008/04.labarta.pdf. 6.2

[138] STMicroelectronics. http://www.st.com/internet/com/home/home.jsp. 2.2.3

[139] Synopsys. Physical Compiler. http://www.synopsys.com/Tools /Implementation/RTLSynthesis/DCUltra/Pages/default.aspx. 2.2.3

[140] Synopsys. PrimeTime Tool. http://www.synopsys.com/Tools /Implementation/SignOff/Pages/PrimeTime.aspx. 2.2.3

[141] T. E. Anderson. The Performance Implications of Spin-Waiting Alternatives for Shared Memory Multiprocessors. In *Proceedings of the Intel Conference on Parallel Processing*, 1989. 4.5

[142] T. Krishna, A. Kumar, L-S. Peh, J. Postman, P. Chiang and M. Erez. Express Virtual Channels with Capacitively Driven Global Links. *IEEE Micro*, 29(4):48–61, 2009. 1.6, 3.2.1, 3.3.1.1, 3.3.2, 4.3.1.1, 4.3.2, 5.4.2, 5.4.3, 5.4.4

[143] T. Krishna, A. Kumar, P. Chiang, M. Erez and Li-Shiuan Peh. NoC with Near-Ideal Express Virtual Channels using Global-Line Communication. In *Proceedings of the 16th IEEE Symposium on High Performance Interconnects*, 2008. 2.4

[144] T. M. Chaves, E. A. Carara and F. G. Moraes. Energy-Efficient Cache Coherence Protocol for NoC-based MPSoCs. In *Proceedings of the 24th Symposium on Integrated Circuits and Systems Design*, 2011. 5.6

[145] Tile-GX Processors. http://tilera.com/products/processors/TILE-Gx_Family. (document), 1.1

[146] V. Krishnan and J. Torrellas. The Need for Fast Communication in Hardware-Based Speculative Chip Multiprocessors. *International Journal of Parallel Programming*, 29(1):3–33, 2001. 3.5

[147] V. Salapura, M. Blumrich and A. Gara. Design and Implementation of the Blue Gene/P Snoop Filter. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture*, 2007. 5.6, 6.2

[148] Verilog. http://www.verilog.com. 2.2.3

[149] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 23(1):20–24, 1995. 1.5

[150] W. Arden, M. Brillouët, P. Cogez, M. Graef, B. Huizing and R. Mahnkopf. Moore-than-Moore, 2010. http://www.itrs.net/Links/2010ITRS/IRC-ITRS-MtM-v2_3.pdf. (document), 1.6, 6.2

[151] W. T.-Y. Hsu and P.-C. Yew. An Effective Synchronization Network for Hot-Spot Accesses. *ACM Transactions on Computer Systems*, 10(3):167–189, 1992. 3.5

[152] W. Zhu, V. C. Sreedhar, Z. Hu and G. R. Gao. Synchronization State Buffer: Supporting Efficient Fine-Grain Synchronization on Many-Core Architectures. In *Proceedings of the 34th International Symposium on Computer Architecture*, 2007. 3.5

[153] X. Chu, R. Buyya, E. Tejedor and R. Badia. A Novel Approach for Realising Superscalar Programming Model on Global Grids. In *Proceedings of the 10th International Conference on High-Performance Computing*, 2009. 6.2

[154] Y. Xu, Y. Du, Y. Zhang and J. Yang. A Composite and Scalable Cache Coherence Protocol for Large Scale CMPs. In *Proceedings of the International Conference on Supercomputing*, 2011. 5.3.3, 5.6

[155] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron and M. Stan. HotLeakage: A Temperature-Aware Model of Subthreshold and Gate Leakage for Architects. Technical Report CS-2003-05, University of Virginia, 2003. 2.2.1

[156] Z. Hu, J. del Cuvillo, W. Zhu and G. R. Gao. Optimization of Dense Matrix Multiplication on IBM Cyclops-64: Challenges and Experiences. In *Proceedings of the 12th International European Conference on Parallel and Distributed Computing*, 2006. 3.5

[157] Z. Huang, X. Shi, Y. Xia and J-K. Peir. Alternative Home: Balancing Distributed CMP Coherence Directory. In *Proceedings of the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2008. 5.6