2010 39th International Conference on Parallel Processing

A *G-line*-based Network for Fast and Efficient Barrier Synchronization in Many-Core CMPs

José L. Abellán Juan Fernández Manuel E. Acacio
Dept. de Ingeniería y Tecnología de Computadores
Facultad de Informática - Universidad de Murcia
30100 Murcia, Spain
{jl.abellan,juanf,meacacio}@ditec.um.es

Abstract

Barrier synchronization in shared memory parallel machines has been widely implemented through busy-waiting on shared variables. However, typical implementations of barrier synchronization tend to produce hot-spots in terms of memory and network contention, thus creating performance bottlenecks that become markedly more pronounced as the number of cores or processors increases. To overcome such limitations, we present a novel hardware-based barrier mechanism in the context of many-core CMPs. Our proposal is based on global interconnection lines (G-lines) and the S-CSMA technique, which have been recently used to enhance a flow control mechanism (EVC) in the context of networks-on-chip. Based on this technology, we have designed a simple and scalable G-line-based network that operates independently of the main data network, and that is aimed at carrying out barrier synchronizations efficiently. In the ideal case, our design takes only 4 cycles to perform a barrier synchronization once all cores or threads have arrived at the barrier. As a proof of concept, we examine the benefits of our proposal by comparing it with one of the best software approaches (a binary combining-tree barrier). To do so, we run several kernels and scientific applications on top of the Sim-PowerCMP performance simulator that models a 32-core CMP with a 2D-mesh network configuration. Our proposal entails average reductions in terms of execution time of 68% and 21% for kernels and scientific applications, respectively. Additionally, network traffic is also lowered by 74% and 18%, respectively.

1 Introduction

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-power 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).

Following the well-known Moore's Law, it is clear that more and more cores will be integrated in future CMP layouts even reaching hundreds of them all integrated in the same chip. In fact, in 2007 Intel introduced a prototype with 80 cores (Intel Polaris), very simple each, but reaching a total peak performance higher than one teraflop. In addition, the latest developments of Intel include a research microprocessor containing 48 cores, intended to scale up to 100 cores and beyond, namely the Single-chip Cloud Computer [1]. CMPs of this kind are commonly referred to as many-core CMPs.

If current trends continue, future many-core CMPs will implement the hardware-managed, implicitly-addressed, coherent caches memory model [14]. With this memory model, all on-chip storage is used for private and shared caches that are kept coherent by hardware. Communication is directly supported by hardware because it occurs implicitly as a result of conventional memory access instructions (i.e. loads and stores). In addition, to guarantee the integrity of shared data structures, synchronization is typically supported by a combination of hardware (through atomic instructions such as *test&set*, *LL/SC* or *fetch&op*), and software (those atomic instructions are used to implement the higher-level mechanisms such as: lock/unlock, for mutual exclusion; and barrier, for global synchronization among threads) [10].

Typical implementations of barrier synchronization are based on a busy-wait process on atomically updated shared variables. However, it is well-known that busy-waiting constitutes one of the major obstacles to scalability [15]. In more depth, busy-waiting involves the CMP's cache coherence protocol which leads to significant levels of contention

0190-3918/10 $26.00 © 2010 IEEE
DOI 10.1109/ICPP.2010.34

at the memory and the interconnection network. Henceforth, barrier synchronizations of this kind will be referred to as software-based barriers.

To overcome the limitations imposed by software-based barrier implementations, there have arisen a lot of proposals that include additional hardware support to perform barrier synchronizations or implement barriers entirely in hardware (see Section 2). In fact, since CMP applications exhibit finer-grained parallelism than conventional multiprocessor applications, a very recent work [17] concludes that barriers for many-core CMPs should be implemented entirely in hardware. Henceforth, barrier synchronizations of this kind will be referred to as hardware-based barriers.

In this paper, we present and evaluate a new hardware-based barrier mechanism in the context of many-core CMPs. Our proposal leverages existing *G-lines* technology and *S-CSMA* technique [27] to deploy a dedicated on-chip *G-line*-based network. The use of a very simple synchronization process over this network provides an extremely efficient implementation for barrier operations. To show the benefits derived from our proposal, we run several kernels and scientific applications on top of a performance simulator (*Sim-PowerCMP*) that models a 32-core CMP with a 2D-mesh network. As we will show, our proposal provides hardware and software simplicity, moreover it meets very-low latency and scalability. For example, Kernel 2 or EM3D achieve reductions of 70% and 54% in execution time, respectively. In addition, since we remove all barrier-related traffic and coherence activity from the interconnection network, network traffic is significantly reduced. For instance, Kernel 2 and EM3D show reductions of 68% and 51% in network traffic, respectively. From this traffic reduction, we believe that our method will also lead to significant improvements in power consumption.

The rest of the paper is organized as follows. Section 2 discusses the related work. We detail our hardware-based barrier mechanism in terms of its architecture and programmability in Section 3. Section 4 describes our simulation environment and presents the benefits of our approach in terms of reductions in execution time and network traffic. Finally, Section 5 presents our main conclusions and plans for future work.

2 Related Work

A variety of hardware-based barrier implementations have been around for a long time. We make an attempt at categorizing some of them in terms of the part of the system they improve or augment. Therefore, we distinguish the following categories: memory-based approaches, network-based approaches, global lines approaches, and special-purpose hardware.

2.1 Memory-based Approaches

The Horizon and Tera MTA [18, 22] use *full/empty* bits in memory at word level to implement synchronized memory references in a cache-less system. To do so, when a processor reads a word its corresponding bit is set to 0 and, when it is written, the bit is set to 1. It implies an efficient synchronization mechanism in terms of a consumer/producer model among processors. Goodman et al. [13] propose a set of efficient primitives for process synchronization based on the use of synchronization bits (syncbits). Syncbits are logically associated with each line in memory to provide a simple mechanism for mutual exclusion. The T3E multiprocessor [24] 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. Monchiero et al. [20] introduce a hardware module to optimize busy-waiting synchronization for 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 network and memory accesses. Zhu et al. [30] propose 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 such as *full/empty* bits based systems do. To do so, the SSB records and manages the states of frequently synchronized data.

2.2 Network-based Approaches

Several proposals are based on modifications of the main data network. For instance, the NYU Ultracomputer [4] uses an enhanced message switching network which combines *fetch&add* synchronization primitives by including adders in the memory network interface. Moreover, Hsu and Yew [29] propose a multistage shuffle-exchange network to efficiently handle synchronization traffic of software barriers by combining packets in the switches in order to relieve hot-spot congestion from the network. Olnowich [11] presents an efficient technique for handling 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 [7] contains a dedicated network (con-

trol 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 [21] also contains a dedicated interconnection network for barrier synchronization.

J. Sartori and R. Kumar [17] indicate that dedicated interconnection networks are the most efficient way to implement barrier synchronizations, at the expense of introducing prohibitive area overheads in some cases. In particular, they analyze the area cost of dedicated links and prove that it may become even worst in the context of CMPs, since die area is a precious resource as it can translate into higher throughput or yield. In fact, the authors propose three barrier implementations, that are hybrid of software and hardware, in order to achieve closer approximation to the performance of a dedicated interconnection network but at a fraction of the cost. These promising proposals use conventional load and store instructions on memory mapped addresses to detect the arrival and completion of the barrier, which implies network traffic overhead in the main data network. However, the authors do not characterize this overhead. On the contrary, our approach does not entail that overhead since it is based on a dedicated *G-line*-based network.

2.3 Global Lines Approaches

Proposals belonging to this category include lines visible to all processors (global lines), used to transfer messages or signals to perform barrier synchronizations. Our hardware-based barrier mechanism could be included in this category, because it also uses global lines to perform barrier synchronizations. The Sequent Balance system [5] 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 semaphores. Krishnan and Torrellas [28] propose a hardware mechanism to support communication and synchronization of registers between on-chip processors for an efficient consumer/producer model. To do so, a Synchronized Scoreboard (SS) is provided per processor. The SSs are connected with a broadcast bus on which register values are transferred. 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 simpler than SLICs, and it is integrated in the context of many-core CMPs.

Shang and Hwang [26] present 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 [19] propose 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. Cyclops [6] is a highly parallel processor-and-memory system on a chip. This architecture implements a fast hardware barrier through a special purpose register (SPR), which is actually implemented as a wired-OR for all threads on the chip. In this way, each thread writes its SPR independently and reads the ORed value of all threads' SPRs waiting for resuming execution. In contrast, our method does not entail any wired-OR or wired-NOR logic to detect the arrivals at the barrier, but it implements a *S-CSMA* technique in a more distributed fashion.

Finally, our proposal is based on *G-lines* technology to deploy a dedicated on-chip *G-line*-based network, and the use of a *S-CSMA* technique. Every *G-line* is basically a shared wire that broadcasts 1-bit messages (signals from now on) across one dimension of the chip in only one clock cycle. *G-lines* enable a *S-CSMA* technique to calculate how many cores are trying to send a message simultaneously. So far, this technology has been used in the context of networks-on-chip (NoC) to enhance a flow control mechanism (EVC) in terms of latency and power consumption [27]. In particular, *G-lines* have been used to broadcast the control signals used in EVC to communicate the availability of free buffers and virtual channels much more accurately (original EVC uses thresholds to communicate available resources conservatively). Furthermore, the authors employ a *S-CSMA* technique to calculate how many virtual channels or free buffers are demanded at any time in order to grant requests accordingly.

2.4 Special-Purpose Hardware

Beckmann and Polychronopoulos [8] present a hardware scheme specifically designed for fast barrier synchronization. This architecture is scalable and supports a large number of concurrent barriers. In more detail, the actual barrier

is a single-bit register BR which is visible to all processors. 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. Sampson et al. [16] present 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.

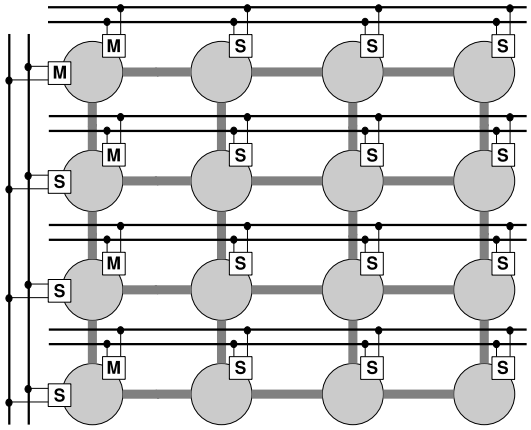

Figure 1. *G-lines* **architecture for a 16-core CMP and 2D-mesh network.**

3 A *G-line*-based Barrier Synchronization

In this section we present our proposal for an efficient and scalable hardware-based barrier synchronization in the context of many-core CMPs. We start by describing the architecture of the *G-line*-based network. As a case study, we choose a CMP with a 2D-mesh data interconnection network, although our proposal is not restricted to this topology. Next, we show how a barrier synchronization would be carried out. We discuss the interface with programmers. And finally, we give details about the implementation of the *G-line* controllers required by our proposal.

3.1 Architecture

Our hardware-based barrier mechanism relies on a *G-line*-based network as can be observed in the example in Figure 1. For simplicity, we concentrate on a version of the proposed network providing support for one barrier. As can be observed, the *G-line*-based network 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 process; and controllers (M and S), that actually implement the synchronization process.

As discussed in Section 2, every *G-line* is a wire that enables the transmission of 1 bit of information across one dimension of the chip in a single cycle. Our proposal needs two *G-lines* per every row and two more for the first column. In this way, for any 2D-mesh layout the total number of *G-lines* per barrier that would be needed is equal to $2 \times (\sqrt{NumCores} + 1)$, where $NumCores$ is the number of cores of the CMP (e.g. 10 *G-lines* for the 16-core CMP shown in Figure 1). It is worth noting that our proposal is aimed at providing this kind of hardware support just for a few number of barriers, which makes that the total amount of *G-lines* that would be required keeps lower than, for example, the 168 used in [27]. Since low power consumption and low area overhead are reported for the latter, we believe that our implementation introduces negligible overhead. Moreover, considering that power consumption in the interconnection network has been recently reported to constitute a significant fraction (approaching 40% in the Raw processor [12]) of the total energy consumed in many-core CMPs, our proposal should bring important savings in terms of energy consumption (we remove all barrier-related traffic and coherence activity from the interconnection network).

In addition of the *G-lines*, our proposal also incorporates a set of controllers. In particular, we distinguish two types of controllers: Master and Slave controllers (see M and S in Figure 1, respectively). Each controller is attached to two *G-lines*: one of them is used to transmit signals, while the other one is employed to receive signals. That is, from the Master standpoint the *G-line* used to receive signals from the Slaves, is used to send signals from the Slaves standpoint, and vice versa. Moreover, the Master controller is responsible for carrying out the count of signals across its *G-line*. To do that, the Master controller contains a device which implements the *S-CSMA* technique. In addition, there are a number of counters and registers such as *Scnt*, *Mcnt*, *flag* and *bar_reg* that will be detailed in the following subsections.

Finally, as in [27], we assume in this work that every *G-line* can support up to six transmitters and one receiver, resulting in a CMP configuration with up to 7×7 cores. In any case, it is worth noting that our hardware-based barrier mechanism could be easily extended to support higher number of cores either by assuming longer latency *G-lines*, or by using groups of *G-line*-based networks linked together through additional *G-lines*.

3.2 Synchronization Process

Next, we describe an example of using our barrier mechanism. Without loss of generality, we assume that all cores execute the same barrier at the same time, and we describe the explanation on a 2×2 mesh layout (see Figure 2). For that, 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 *G-lines* 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). Furthermore, Figure 2 also shows two types of counters belonging to each Master controller: *ScntH* or *ScntV*, which stores 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) attached to the horizontal or vertical *G-lines*, respectively; and *Mcnt*, that stores 1 if the corresponding core (0 or 2) arrives at the barrier, and 0 otherwise. Notice that, *Scnt* counters would not be necessary in this particular case, because only one Slave transmit signals as much. Nevertheless, we include them for the sake of completeness.

The process is as follows. At cycle 0, the horizontal Slaves (Sh) signal, through their corresponding transmission *G-lines*, the arrival of cores 1 and 3 at the barrier and wait until their horizontal Masters (Mh) command to resume execution, by monitoring the reception horizontal *G-lines*. Then, the horizontal Masters count the number of received signals and update their counters *ScntH*=1 (because there is just one Slave for each), besides they also set the counter *Mcnt* to 1 due to the arrival of cores 0 and 2 at the barrier. At cycle 1, upon each horizontal Master updates both counters to its maximum values, its corresponding vertical Slave (Sv) repeats the process. In particular, the vertical Slave writes into its *G-line*, and the vertical Master (Mv) updates the counter *ScntV*=1, but also sets the counter *Mcnt* to 1 because their corresponding horizontal Master (in core 0) have updated its *Mcnt*=1. At cycle 2 the release stage starts in order to resume execution by using the unused *G-lines*. To do so, the vertical Master signals its vertical Slave and resets all counters. Finally at cycle 3, the horizontal Masters do the same on their corresponding horizontal *G-lines* to wake up the horizontal Slaves. We will give a more detailed explanation in Subsection 3.4.

3.3 Programmability Issues

A very important objective for us was to provide programmers the mechanism in the simplest possible way. To do so, we deal with both the programmer and architectural points of view. Regarding the former, the use of our barrier mechanism is as simple as illustrated in Figure 3. Programmers should use an special register, called *bar_reg*, to notify the arrival at the barrier (by assigning it a value greater than zero). Then, each core enters in a loop waiting for the rest of cores. Once all cores have done the same, our hardware mechanism resets all registers *bar_reg*, and all cores can resume execution. Finally, regarding the architectural standpoint, we augment the register file of each core with the *bar_reg* register and enable the interplay between controllers and these registers. That is, switching on the controllers, when the registers are assigned, and once all controllers have finished the synchronization (cycle 3 in Figure 2), resetting the registers and switching off the controllers. In this way, we also limit the power consumed by the controllers.

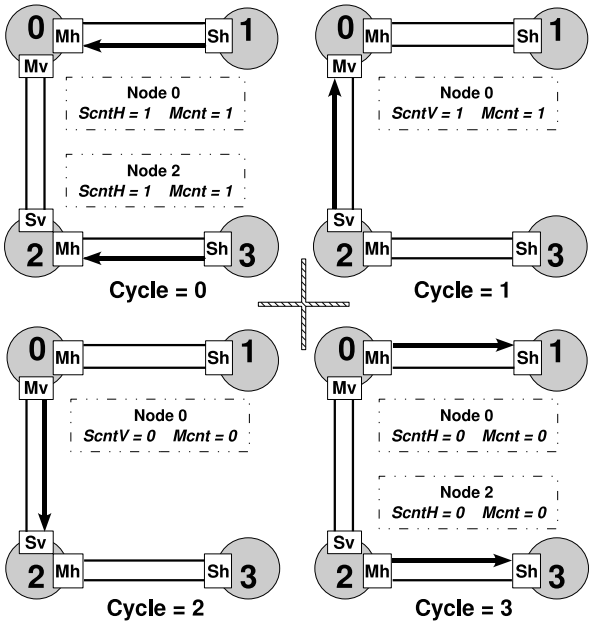

Figure 2. Steps for our barrier synchronization.

```
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. Programming our barrier mechanism.

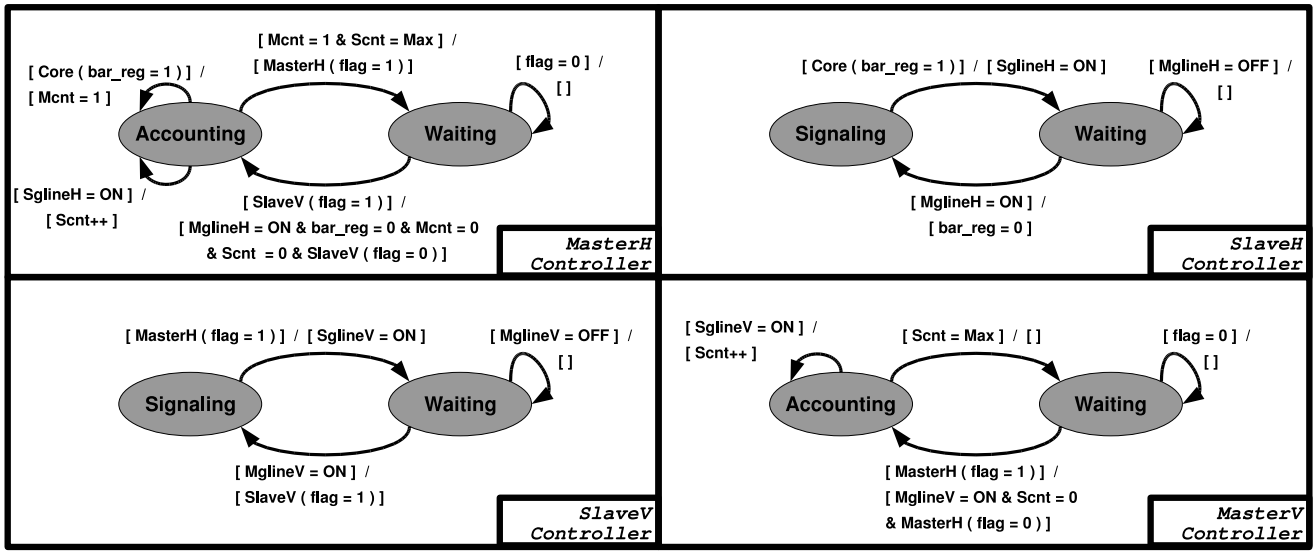

Figure 4. Finite State Automata that implement the *G-line* controllers.

3.4 *G-line* Controllers Implementation

Finally, we take a closer look at the implementation of the *G-line* controllers in order to have a more clear understanding of the above subsections (see Figure 4). As we can see, there are four 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 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]. In more depth, we distinguish the following events and actions:

- A core writes the register *bar_reg*: *Core(bar_reg=1)*.
- A *G-line* controller writes a *G-line*: *XglineY=ON*, where X identifies the controller (M for Master, and S for Slave), and Y identifies the *G-line* (V or H, for Vertical or Horizontal *G-lines* respectively).
- Update of registers: *Mcnt=1*, or *X(flag=1)*, where X identifies the controller that updates the register in parenthesis. This is treated as an special case in which the controller waits for an specific value of the register updated by the other controller belonging to the same core. For example, in the transition from the state *Waiting* to the state *Accounting* of the MasterV automaton, there is an event *MasterH(flag=1)*. It means that the MasterV controller has to perform that transition when the MasterH controller sets the register to 1, both belonging to the same core (e.g. the core 0 in Figure 2).
- Finally, no event or action: *[]*.

The SlaveH begins in the state *Signaling* waiting for the arrival of the corresponding core at the barrier. Upon arrival, the core writes its register *bar_reg* which triggers the event *Core(bar_reg=1)* and performs the action *SglineH=ON*, because this horizontal slave controller writes into the horizontal *G-line*. Then, the automaton switches to the *Waiting* state until the release phase is reached. The MasterH begins in the state *Accounting* in order to count the signals from the SlaveH controllers in the horizontal *G-line* (the event *SglineH=ON*), and then updates the register *Scnt* (the action *Scnt++*) properly. Notice that, through the *S-CSMA* technique is possible to account for more than one signal at the same cycle. Moreover, in this state the controller waits until the corresponding core arrives at the barrier. Upon the arrival, the core writes its register *bar_reg* which triggers the event *Core(bar_reg=1)* and performs the action *Mcnt=1*. Next, once all SlaveH controllers have signaled through the *G-line* SglineH, and the register *Mcnt* is set to 1, the action *MasterH(flag=1)* is performed. Finally, the automaton switches to the *Waiting* state until the release phase is reached. The SlaveV begins in the state *Signaling* until the register *flag* is set to 1 by the MasterH controller (the event *MasterH(flag=1)*). Once the last condition is satisfied, the SlaveV writes into the vertical *G-line* (the action *SglineV=ON*). Then, the automaton switches to the *Waiting* state until the release phase is reached. Finally, the MasterV begins in the state *Accounting* in order to count the signals from the SlaveV controllers in the vertical *G-line* (the event *SglineV=ON*), and update the register *Scnt* properly (the action *Scnt++*). Next, once all SlaveV controllers have signaled through the *G-line* SglineV (the event *Scnt=Max*), the automaton switches to the *Waiting* state until the release phase is reached. Finally, upon MasterH sets

Table 1. 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, 1 cycle
L2 Cache (per core)	256KB, 4-way, 6+2 cycles
Memory access time	400 cycles
Network configuration	2D-mesh
Network bandwidth	75 GB/s
Link width	75 bytes

the register *flag* to 1, the release phase starts which implies all the transitions from the right state to the left state in all the automata, following the order: MasterV, SlaveV, MasterH and SlaveH.

4 Evaluation

In this section we present the details of our experimental methodology and performance results. We describe the simulation environment in Section 4.1 and the set of benchmarks that we have used in Section 4.2. The performance results are given in Section 4.3.

4.1 Testbed

We have extended Sim-PowerCMP [3] performance simulator to support our hardware-based barrier mechanism. In short, Sim-PowerCMP is a detailed architecture-level power-performance simulation tool that simulates tiled-CMP architectures with a shared L2 cache on-chip and a directory-based cache coherence protocol. Moreover, our CMP configuration has up to 32 cores and an aggressive 2D-mesh network. Table 1 summarizes the values of the main configurable parameters assumed in this work.

4.2 Benchmarks

In order to estimate performance improvements, we have considered the following benchmarks: a synthetic benchmark, various kernels from Livermore Loops [2] (Kernels 2, 3 and 6); and three scientific applications: OCEAN, that belongs to the SPLASH-2 benchmark suite [23], UNSTRUCTURED [25], and finally, EM3D [9]. The set of benchmarks is summarized in Table 2. As we can see, we illustrate for each one: the input size, the total number of barriers (*#Barriers*), and the estimated barrier period (the number of cycles in average between two consecutive barrier executions). The latter is calculated by dividing the total number of execution cycles by the total number of barriers in every case.

The synthetic benchmark is intended to measure the latency of barriers themselves. Hence, it helps us provide some insight into the benefits obtained by our approach. To do that, we follow the methodology described in [10]: performance is measured as average time per barrier over a loop of four consecutive barriers with no work or delays between them, with the loop being executed 100,000 times. Livermore loops 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 is hard to extract and exploit efficiently. By the same recommendations given in [16], we focus on Kernels 2, 3 and 6. In short, Kernel 2 is an excerpt from an incomplete Cholesky conjugate gradient code. Kernel 3 is a simple inner product. Finally, Kernel 6 is a general linear recurrence equation. Regarding the applications used, UNSTRUCTURED is a computational fluid dynamic application; EM3D is a shared memory implementation of the Split-C benchmark; and, OCEAN studies large-scale ocean movements based on eddy and boundary currents. We would like to point out that OCEAN is the application with more barrier executions from the SPLASH-2 benchmark suite [23]. However, it presents in average one barrier for every 205,206 cycles (see Table 2). This high barrier period that SPLASH-2 features has been already discussed in other works such as [16].

4.3 Performance Results

To obtain the performance results, we compare our hardware-based barrier (GL from now on) with two software-based implementations. On the one hand, a centralized sense-reversal barrier based on locks (or CSW), where each core increments a centralized shared counter as it reaches the barrier, and spins until that counter indicates that all cores are present. On the other hand, a binary combining-tree or distributed barrier (DSW), where 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 this scenario, 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 approach should improve all three stages because they are executed without involving any network transaction, but just by means of operations on a *G-line-based* network taking only 4 cycles in the best case. How-

Table 2. Configuration of the benchmarks used in this work.

Benchmark	Input Size	#Barriers	Barrier 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	2,862
Kernel 6	1,024 elements, 1,000 iterations	1,022,000	4,908
OCEAN	258x258 ocean	364	205,206
UNSTRUCTURED	Mesh.2K, 1 time step	80	67,361
EM3D	38,400 nodes, degree 2, 15% remote, 25 time steps	198	3,673

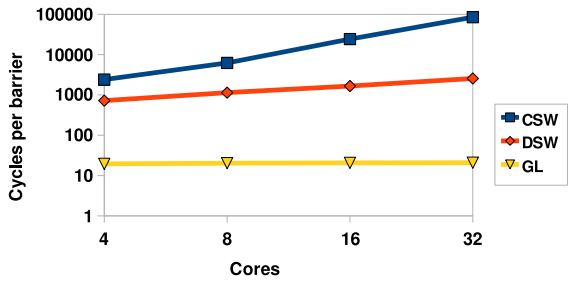

Figure 5. Average times for three different barrier mechanisms.

ever, we can identify two typical situations in which our approach may entail negligible improvement: applications containing a reduced number of barriers; and when barrier latency is dominated by the stage S2. The former helped us to pick the most significant benchmarks for our proposals (e.g. OCEAN from SPLASH-2 benchmark suite). The latter may suggest that the application is under workload imbalance. We will take these conclusions in mind when analyzing the performance results in the two following subsections.

4.3.1 Execution Time

As explained above, to measure the latency of barriers themselves, we use the synthetic benchmark. From the results presented in Figure 5, we can derive two appreciations. First, the distributed software-based barrier is much more efficient than the centralized software-based barrier. And second, it is clear that our mechanism highly outperforms the others in both efficiency and scalability. However, we noticed an slight overhead in the times obtained for GL: 13 cycles instead of the theoretical 4 cycles (see Figure 2). This 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 6 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 software-based barrier implementation (DSW) and our hardware barrier mechanism (GL). Execution time is further broken down into several categories: *Barrier* is the time spent in barriers (sum of the time taken in the S1, S2 and S3 stages explained above); *Write* and *Read* are the times spent for operations on memory; *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 *AVG_K* and *AVG_A*).

Regarding the kernels results, we can see that our proposal involves a reduction in execution time of 68% (see *AVG_K*). In more depth, Kernels 2, 3 and 6 present reductions of 70%, 88% and 47%, respectively. These reductions stem from the barrier period described in Table 2 (3,103, 2,862 and 4,908 cycles, respectively). Since our mechanism does not need any load or store instruction on memory to perform the barrier synchronization, the kernels also present a reduction in the categories *Write* and *Read*. Besides, although Kernel 3 shows 4% in column DSW for the category *Write*, its code [2] does not contain any instruction that motivates that value (its iterations do not contain any store instruction on memory). It means that this is due to the overhead introduced by the software-based barrier implementation (DSW). As a result, our mechanism reduces to zero that category. In addition, Kernel 3 presents a barrier period almost equal to the ideal case (the synthetic benchmark): 2,862 cycles vs. 2,568 cycles. That explains its huge reduction in execution time.

On the contrary, the applications show a lesser reduction in execution time of only 21% (see *AVG_A*). This is due to the very high barrier period of the scientific applications UNSTRUCTURED and OCEAN (67,361 and 205,206 cycles, respectively). As a result, it implies reductions of only 3% and 5% respectively. On the other hand, significant reductions in execution time (54%) are observed for EM3D due to its very low barrier period (3,673 cycles).

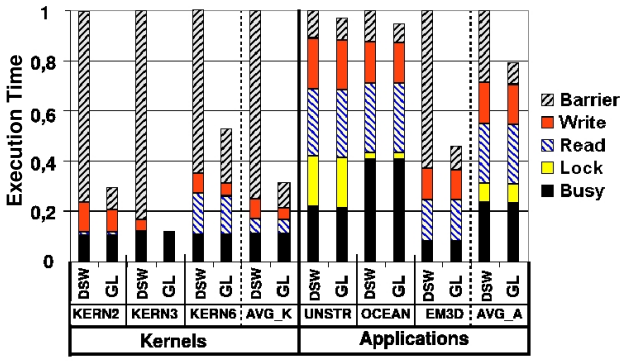

Figure 6. Average normalized times for the benchmarks used in this work over a 32-core CMP.

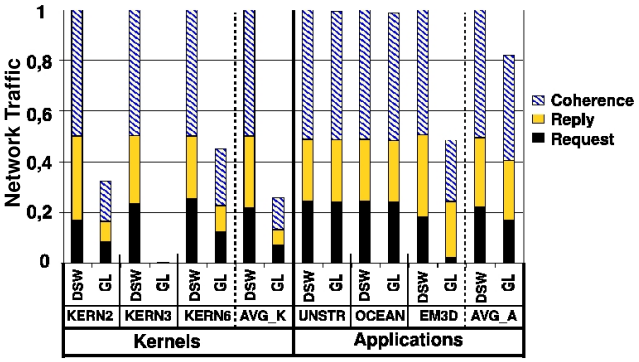

Figure 7. Average normalized number of messages across the network for the benchmarks used in this work over a 32-core CMP.

4.3.2 Network Traffic

As aforementioned, our proposal entails neither coherence activity nor barrier traffic in the main interconnection network. Therefore, we analyze the network traffic over the main data network for the same experiments as before. Then, Figure 7 depicts the average normalized number of messages across the network. Moreover, each column is broken down into three categories: *Coherence* is the traffic generated by the coherence protocol; *Request* stems from store or load operations to send or receive data from the system memory, respectively; finally, *Reply* involves the traffic which contains the data requested by the last category.

Regarding the kernels, notice that there is an important reduction in network traffic (see 74% in *AVG_K*) according to the reductions in time for the categories *Barrier*, *Write* and *Read* presented in the last subsection. We would like to point out the vast reduction in network traffic for Kernel 3 (99.82%) because, as we mentioned in Subsection 4.3.1, almost all the traffic generated in this benchmark is due to the barrier.

Finally, regarding the scientific applications, we can see a slight reduction in network traffic (see 18% in *AVG_A*). It is explained from the negligible reductions in time for the categories *Barrier*, *Write* and *Read* given for the applications UNSTRUCTURED and OCEAN in the last subsection. We could expect more than 1% reduction in network traffic for both applications, due to the 3% and 5% reduction in execution time, respectively. However, we noticed that the latency of barriers is dominated by the S2 stage and, as we mentioned, this implies workload imbalance. In DSW, this stage involves negligible network traffic because, once shared variables are loaded in cache, busy-waiting is performed locally. As a consequence, our method reports this low traffic reduction. Finally, as we expected, EM3D presents a considerable reduction in network traffic (51%) because of its low barrier period.

5 Conclusions and Future Work

We have presented a novel hardware-based barrier mechanism for many-core CMPs. Our architecture has not any influence on the conventional memory system to conduct the synchronization, since we remove all coherence activity and barrier-related traffic from the interconnection network. Instead, it relies on one-bit messages across a *G-line*-based network that interconnects all cores, which is independent of the main data network, along with the use of the *S-CSMA* technique. We have measured the benefits of our approach by means of several benchmarks running on top of Sim-PowerCMP: a synthetic benchmark, kernels 2, 3 and 6 from Livermore loops suite, and three scientific applications: OCEAN, UNSTRUCTURED and EM3D. In light of our performance results, we would like to point out that our mechanism meets efficiency in terms of execution time and reduction of traffic in the main data network. For instance, the kernels and the scientific applications under study show average reductions of 68% and 21% in total execution time, and 74% and 18% in network traffic, respectively. Moreover our proposal meets hardware and software simplicity because of the little and scalable hardware logic needed, and due to its easy programmability.

As future work, we will extend the use of our approach to a more general context by multiplexing in space and time, in which several barrier executions can coexist. Moreover, we will design efficient and scalable schemes to interconnect *G-line*-based networks, in order to overcome the limitation in the number of cores supported by this technology (a many-core CMP with more than 7×7 2D-mesh). Finally, we will measure the efficiency of our method in terms of power consumption.

Acknowledgments

This work was supported by the Spanish MEC and MICINN, as well as European Comission FEDER funds, under Grants "CSD2006-00046" and "TIN2009-14475-C04". José L. Abellán is supported by fellowship 12461/FPI/09 from Comunidad Autónoma de la Región de Murcia (Fundación Séneca, Agencia Regional de Ciencia y Tecnología).

References

[1] http://techresearch.intel.com/articles/Tera-Scale/1826.htm.

[2] http://www.netlib.org/benchmark/livermorec.

[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 21th International Conference on Advanced Information Networking and Applications Workshops*, 2007.

[4] A. Gottlieb et al. The NYU ultracomputer – designing a MIMD, shared-memory parallel machine. In *Proceedings of 9th Annual International Symposium on Computer Architecture*, 1982.

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

[6] C. Cascaval et al. Evaluation of a Multithreaded Architecture for Cellular Computing. In *Proceedings of 8th International Symposium on High-Performance Computer Architecture*, 2002.

[7] C. E. Leiserson et al. The Network Architecture of the Connection Machine CM-5. In *Proceedings of ACM Symposium on Parallel Algorithms and Architectures*, 1992.

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

[9] D. E. Culler et al. Parallel Programming in Split-C. In *Proceedings of International SC High Performance Networking and Computing Conference*, 1993.

[10] D. E. Culler, J. P. Singh and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1998.

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

[12] H. Wang, L-S. Peh and S. Malik. Power-driven Design of Router Microarchitectures in On-chip Networks. In *Proceedings of 36th IEEE/ACM International Symposium on Microarchitecture*, 2003.

[13] J. Goodman, M. K. Vernon and P. J. Woest. Efficient synchronization Primitives for large-scale cache-coherent shared-memory multiprocessors. In *Proceedings of 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 1989.

[14] J. Leverich et al. Comparing Memory Systems for Chip Multiprocessors. *ACM SIGARCH Computer Architecture News*, 35(2):358–368, 2007.

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

[16] J. Sampson et al. Exploiting Fine-Grained Data Parallelism with Chip Multiprocessors and Fast Barriers. In *Proceedings of 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[17] J. Sartori and R. Kumar. Low-Overhead, High-Speed Multicore Barrier Synchronization. In *Proceedings of 5th International Conference on High Performance Embedded Architectures and Compilers*, 2010.

[18] J. T. Kuehn and B. J. Smith. The Horizon Supercomputing System: Architecture and Software. In *Proceedings of International Conference on Supercomputing*, 1998.

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

[20] M. Monchiero et al. An efficient synchronization technique for multiprocessor systems on-chip. *ACM SIGARCH Computer Architecture News*, 34:33–40, 2006.

[21] P. Coteus et al. Packaging the Blue Gene/L Supercomputer. *IBM Journal of Research and Development*, 49:213–248, 2005.

[22] R. Alverson et al. The Tera Computer System. In *Proceedings of 4th International Conference on Supercomputing*, 1990.

[23] S. C. Woo et al. The SPLASH-2 programs: Characterization and Methodological Considerations. In *Proceedings of 22nd International Symposium on Computer Architecture*, 1995.

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

[25] S. S. Mukherjee et al. Efficient Support for Irregular Applications on Distributed-Memory Machines. In *Proceedings of 5th International Symposium on Principles and Practice of Parallel Programming*, 1995.

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

[27] T. Krishna et al. NoC with Near-Ideal Express Virtual Channels using Global-Line Communication. In *Proceedings of 16th IEEE Symposium on High Performance Interconnects*, 2008.

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

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

[30] W. Zhu et al. Synchronization State Buffer: Supporting Efficient Fine-Grain Synchronization on Many-Core Architectures. In *Proceedings of 34th Annual International Symposium on Computer Architecture*, 2007.