A Novel Hardware-based Barrier Synchronization for 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

We present in this work a novel hardware-based barrier mechanism for synchronization on many-core CMPs. In particular, we leverage global interconnection lines (G-lines) and S-CSMA technique, which have been used to overcome some limitations of a flow control mechanism (EVC) in the context of Networks-on-Chip, we propose a simple G-line-based network that operates independently of the main data network in order to carry out barrier synchronizations. Next, we evaluate our approach by running several applications on top of the Sim-PowerCMP performance simulator. Our method only takes 4 cycles to carry out the synchronization once all cores or threads have arrived at the barrier. Hence, we obtain much better performance results than software-based barrier implementations in terms of scalability and efficiency.

1 Introduction

Nowadays, Chip-multiprocessors (or CMPs) constitute the best way to take advantage of the increasing number of transistors available in a single die, since they provide higher-performance and lower-power than complex uniprocessor systems exploiting thread-level parallelism (TLP). A myriad of CMPs have arisen in almost all market segments: embedded systems (Cell Broadband Engine), personal computers (Intel Core2 Extreme Quad-core), servers (Sun UltraSPARC T2) and even as building blocks in supercomputers (IBM BlueGene/P).

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. CMPs of this kind are commonly referred to as many-core CMPs. 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.

To extract high performance from such architectures, applications must be divided into multiple threads which should be executed on separate cores in order to increase concurrency and exploit TLP. Moreover, threads need to communicate and synchronize among them, thus hardware and software support is strictly necessary for programmers to develop their parallel applications. Currently, most of these CMP designs implement a shared-memory programming model. Communication is directly supported by hardware because it occurs implicitly as a result of conventional memory access instructions (i.e. loads and stores). On the other hand, synchronization is typically supported by a combination of hardware (through atomic instructions from the specific core's ISA: *test&set*, *LL/SC* or *fetch&op*), and software (those atomic instructions are used to implement the higher-level mechanisms such as: locks/unlocks, for mutual exclusion; and barriers, for global synchronization among threads). In fact, these implementations typically imply busy-waiting on shared variables which is actually the major obstacle to scalability [4]. In more depth, busy-waiting involves the CMP's coherence protocol which tends to introduce significant levels of contention at the memory and the interconnection network. Therefore, it introduces performance bottlenecks that become markedly more pronounced as the number of cores increases. Henceforth, busy-waiting based barrier synchronization will be referred to as software-based barriers.

On the other hand, hardware-based barriers have arisen in the context of multiprocessors in order to overcome the limitations given by software-based implementations. Conceptually, a single wired-AND global line is enough to perform the synchronization: a processor sets its input high when it reaches the barrier and waits until the output goes high before it can proceed. So far, a global line has not been integrated in CMPs due to technical issues. However, the latest developments in silicon technology show the possibility of including *global interconnection lines* (*G-lines* from now on) in future CMP designs by attaching 7 cores as much [6]. In more detail, these *G-lines* provide a one cycle broadcast across the chip between one transmitter and one

receiver on a per-bit granularity. In addition, one receiver is able to receive signals from more than one transmitter at the same time. In turn, a technique referred to as *smart carrier sense multiple access* (*S-CSMA*) is enabled for the receiver to sense the number of transmitters utilizing the line at any given clock cycle. However, this new architecture has been used in the context of networks-on-chip (NoC) [6] to enhance a flow control mechanism (EVC) in terms of latency and power consumption.

Our main contribution in this work is that we leverage the above technology (*G-lines* and *S-CSMA* technique) to develop a hardware-based barrier in the context of many-core CMPs. As we will discuss, our proposal meets hardware and software simplicity (in Section 2), moreover it meets very-low latency and scalability (in Section 3). Finally, since we remove all barrier-related traffic and coherence activity from the interconnection network, we believe that it will reduce power consumption and hot-spots in the main interconnection network (as future work).

2 Hardware-based Barrier

In this section, we present our hardware-based barrier mechanism.

2.1 Architecture

As already discussed, our approach do not have any influence on the memory system of the CMP. It does not need any shared variables, it does not entail neither coherence activity nor barrier traffic in the main interconnection network. Therefore, it unloads an important overhead from the memory system. Instead, it relies on 1-bit messages across a *G-line*-based network that interconnects all cores, along with the use of the *S-CSMA* technique. It is graphically outlined in Figure 1 for a 4x4 mesh network. Notice that, as opposed to the 14 *G-lines* per direction per row and column proposed in [6], we only use two *G-lines* per every row and two more for the first column. In general, our approach only needs for any 2D-mesh (7x7 mesh as much, as explained in Section 1) : $2x(\sqrt{NumCores}+1)$ *G-lines*.

As we can see, the *G-line*-based network interconnects all cores through two sort of controllers, namely Master (M) and Slave (S). Each controller is attached to two *G-lines*: one of them is used to transmit signals and the other one for receiving 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*. It performs the accounting by means of a device which implements the *S-CSMA* technique. In addition, there are a number of counters and registers such as *Scnt*, *Mcnt* and *bar_reg* that will

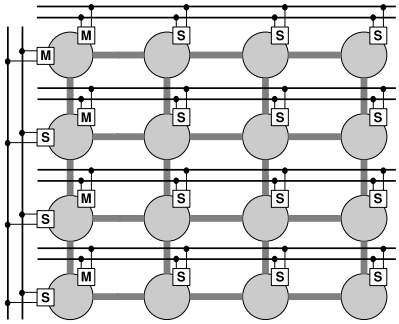

Figure 1. *G-lines* **architecture for a 16-core CMP and 2D-mesh network.**

be detailed in the following subsections in order to have a clear understanding of them.

2.2 Synchronization Process

Next, we explain the process of barrier synchronization for any given many-core CMP with a 2D-mesh network (see Figure 2). Without loss of generality, we assume that all cores execute the same barrier at the same time, and we describe the explanation on a 2x2 mesh layout. To clarify the explanation, 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, the Figure also shows two types of counters belonging to each Master controller: *Scnt*, which stores the number of signals (obtained through the *S-CSMA* technique), received from the Slaves (cores 1 or 3) attached to the *G-line*; and *Mcnt*, that stores 1 if the corresponding core (0 or 2) arrives at the barrier, and it stores 0 otherwise.

The process is as follows. At cycle 0, the horizontal Slaves (Sh) signal, through its corresponding *G-line*, the arrival of cores 1 and 3 at the barrier and wait until their horizontal Masters (Mh) command to resume execution, by monitoring the other *G-lines*. Then, the horizontal Masters count the number of received signals and update their counters *Scnt*=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 *Scnt*=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 us-

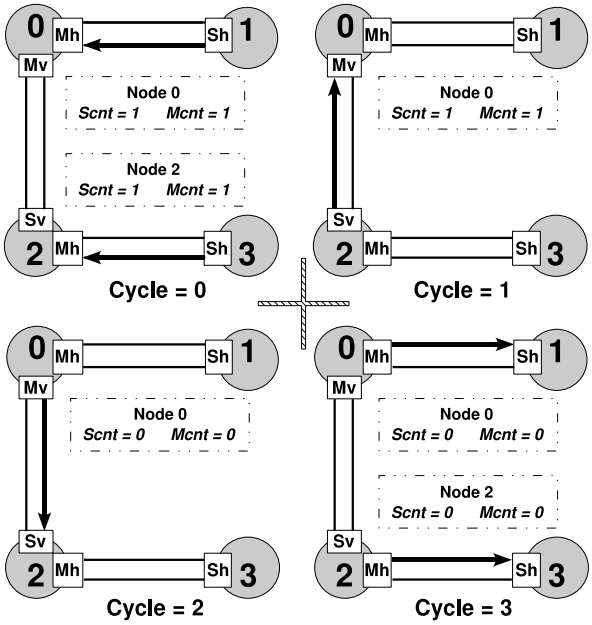

Figure 2. Steps for our barrier synchronization.

ing 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.

2.3 Programmability Issues

A very important purpose for us was to provide programmers the mechanism in the simplest way as possible. To do so, we dealt with the programmer and architectural point of view. Regarding the former, the use of our barrier mechanism is as simple as illustrated in Figure 3. Programmer 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* and enable the interplay between controllers and these registers. That is, switch on the controllers, when the registers are assigned, and once all controllers have finished the synchronization (cycle 3 in Figure 2), reset the registers and switch off the controllers. In this way, we also limit the power consumed by the controllers.

3 Evaluation

As testbed, we have extended Sim-PowerCMP [2], a detailed architecture-level power-performance simulation tool

```
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.

Table 1. CMP baseline configuration.

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 (8X-B-Wires)

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 a 2D-mesh network (for further details see Table 1).

As benchmarks, we consider a synthetic benchmark and various kernels from Livermore Loops [1]. Regarding the former, it help us provide some insight into the benefits obtained by our approach in an ideal scenario. Furthermore, it follows the methodology described in [3]. Thus, 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. Regarding the latter, 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 [5], we focus on following kernels: Kernel 2, which 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.

In the literature there are a lot of algorithms for software-based barriers. As we will discuss in Section 3, we focus on two of them [4]. On the one hand, a centralized sense-reversal barrier based on locks (or CSW), where each core increments a centralized shared count as it reaches the barrier, and spins until that count indicates that all cores are present. On the other hand, a binary combining-tree or distributed barrier (DSW), where there are several shared

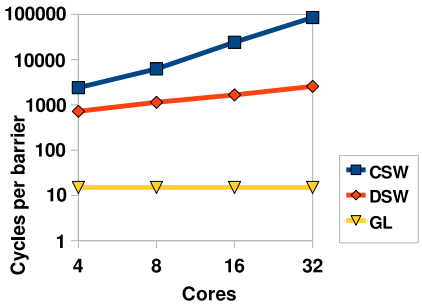

Figure 4. Average times for three different barrier mechanisms.

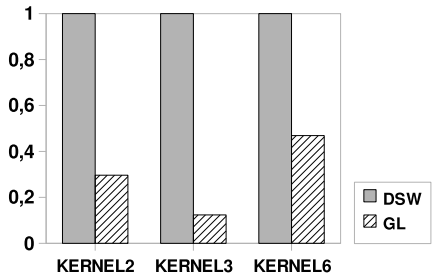

Figure 5. Average normalized times for Kernels 2, 3 and 6 over a 32-core CMP.

counters distributed in a binary tree. 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 sense (towards the leaves).

From the results presented in Figure 4, we can derive two appreciations. First, the very efficient distributed software-based barrier as opposed to the centralized software-based barrier. And second, it is clear that our mechanism highly outperforms the others in both efficiency and scalability. However, we appreciated an slight overhead in GL times: 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.

Finally, Figure 5 shows the average normalized times for all kernels executed over a 32-core CMP layout (1,000 iterations and 1,024 elements). As we can see, comparing our approach and the best based-software barrier implementation (DSW), our hardware barrier mechanism reports much better performance than the software barrier. In more depth, we obtained the percentage of barrier time (through the statistics reported by Sim-PowerCMP) for each kernel executed with the DSW version: 81% (K2), 89% (K3) and 61% (K6). As a result, the improvements of our approach are clearly justified.

4 Conclusions and Future Work

In this 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: no shared variables, no coherence traffic and no traffic across the interconnection network. Instead, it relies on one-bit messages across a *G-line*-based network that interconnects all cores, along with the use of the *S-CSMA* technique. We have measured the benefits of our approach by means of several applications running on top of Sim-PowerCMP: a synthetic benchmark and kernels 2, 3 and 6 from Livermore loops suite. In light of our performance results, we would like to point out that our mechanism not only meets scalability, but also meets efficiency because, once all cores arrive at the barrier, only four cycles are needed to perform the synchronization process. Moreover it 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 measure the efficiency of our method in terms of power consumption and quantify the reduction in terms of network traffic. Moreover, we will multiplex in space (virtual hierarchies) and time the use of our hardware-based barrier in many-core CMPs.

Acknowledgments

This work has been jointly supported by the Spanish MEC and European Commission FEDER funds under grants “Consolider Ingenio-2010 CSD2006-00046” and “TIN2006-15516-C04-03”. 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://www.netlib.org/benchmark/livermorec.

[2] A. Flores et al. Sim-PowerCMP: A Detailed Simulator for Energy Consumption Analysis in Future Embedded CMP Architectures. In *Proceedings of the 21th International Conference on Advanced Information Networking and Applications Workshops*, May 2007.

[3] D. E. Culler et al. Parallel Computer Architecture: A Hardware/Software Approach. August 1998.

[4] J. M. Mellor-Crummey et al. Synchronization without Contention. *ACM SIGARCH Computer Architecture News*, April 1991.

[5] 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*, December 2006.

[6] 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*, August 2008.