NoC-based thread synchronization in a custom manycore system

Alessandro Cilardo, Mirko Gagliardi, Daniele Passaretti

Abstract This workshop paper presents an efficient hardware support for thread synchronization in a customized manycore system developed within the MANGO H2020 project. The solution relies on a distributed master and on a lightweight control unit on the core side, using hardware-level messages and thus avoiding memory accesses. It supports multiple barriers for different application kernels executed simultaneously. The results for different NoC sizes provide indications about the reduced synchronization times and the area overheads incurred by our solution.

1 Introduction

The increasing need for resource- and power-efficient computing over the last decade has stimulated the emergence of compute platforms with moderate or high levels of parallelism, like GPU, SIMD, and manycore processors in a variety of application domains [1, 2, 10, 5]. In particular, manycore systems are based on a considerable number of lightweight processor cores typically connected through a Network-on-Chip (NoC) [3, 23], providing a scalable approach to the interconnection of parallel on-chip systems. In fact, on-chip connectivity has been attracting much interest during the last years, also including recent developments at the physical technology level [6, 8]. While early NoCs, like the Epiphany mesh-based interconnect [14], used a flat cache-less memory model, caching and related coherence management has become a crucial feature in today's NoCs needed to improve performance and preserve programmability in manycore systems. Examples of modern manycore solutions include an integrated 80-tile NoC prototype architecture, based on an on-chip 2D mesh topology, proposed by Intel [12], and the Tilera TILE64 processor [11], based on three-wide VLIW compute cores with 64-bit instruction

Alessandro Cilardo, Mirko Gagliardi, Daniele Passaretti
University of Naples Federico II / CeRICT, via Claudio 21, 80125, Napoli, Italy, e-mail: acilardo@unina.it

words as well as a scalable 2D mesh network with support for coherent shared memory, where each core can directly access any other cache through the interconnect. The Tilera NoC infrastructure in fact provides five different networks for different uses, including one dedicated to memory transactions. Being targeted at parallel applications, these systems are expected to provide some form of support for thread synchronization, e.g. *barrier* primitives. Existing solutions use a variety of hardware- or software-based techniques. The work in [7] describes some synchronization algorithms implemented in software, that rely on a message passing infrastructure and different NoC topologies. In [24] the authors implemented a dedicated G-line net dedicated to barriers. This solution is limited to ASIC architectures and requires a wire for each tile. In [25], the authors support thread synchronization in a packet-switched manycore NoC by relying on two types of links.

This work presents an activity developed in the context of the *MANGO: exploring Manycore Architectures for Next-GeneratiOn HPC systems* H2020 research project. As part of the manycore architecture exploration carried out by the project, the authors developed a customizable GPU-like core used as the compute element in a configurable NoC with support for coherent shared memory. The manycore system is tested on a large-scale FPGA platform developed by MANGO, where hardware reconfigurability is mostly used for emulation purposes, although FPGAs could also be chosen as the final acceleration technology for certain classes of workloads [4, 9, 16, 19]. This workshop paper specifically explores the adoption of a distributed and NoC-based synchronization mechanism. At the heart of our approach is a distributed synchronization master inspired by the directory-based coherence protocol. Relying on the distributed approach as well as the lightweight three-staged synchronization client on the core side, the proposed architecture can support multiple synchronizations for different application kernels running concurrently. The paper describes the main insights in our approach, the resulting architecture and the way it handles multiple synchronizations concurrently, as well as the advantages of the distributed mechanism.

The rest of the paper is organized as follows. In Section 2 we summarize previous techniques for supporting barrier synchronization in manycore architectures. In Section 3 we describe the baseline manycore system as well as the synchronization hardware introduced by this work. In Section 4 we describe the implementation of the barrier function, while in Section 5 we evaluate it with a synthetic benchmark both in a central and distributed configuration for different NoC sizes.

2 Related work

Barrier synchronization is a common primitive used to separate in time different phases of a parallel application. Efficient support for barrier synchronization in manycore systems is of paramount importance because of its role in parallel code. Culler and Gupta in [22] address this problem for shared-memory multiprocessors. They present a software solution, which relies on hardware atomic instructions and

memory coherence. The solution uses a lock variable and a counter keeping track of all the cores that have hit the synchronization point. Indeed, the literature offers various software-based barrier solutions. Hoefler and Rehm [7] present a study on software barrier algorithms combining both shared memory and message passing techniques. The authors implement typical barrier synchronization algorithms, such as Butterfly and Combining tree, and analyze message overheads and the required memory for each of them. The results in [18] show that software barriers, even when based on hardware message passing, incur considerable overheads causing NoC resources to stay underutilized, unlike hardware barriers. As a consequence, many NoC-based techniques have been proposed in the last years. In [24], the authors implement a hard-wired barrier mechanism called G-line net. Each core has a dedicated wire connected to the barrier master. Synchronization is established as soon as each core asserts its line. This technique uses multidrop connectivity and the S-CSMA collision detection in order to provide a flow control mechanism (EVC), enhancing performance in terms of latency and power consumption. Such a solution does not require memory accesses, and provides fast synchronization compared to software solutions. However, the approach may lack scalability and supports only one barrier at a time. The authors of [21] propose a communication unit in a NoC-based system, relying on a barrier controller and a communication unit that enable synchronization operations. The control unit is integrated into a general NoC switch and communicates with a centralized master located in the network. The paper proposes an efficient control mechanism, but it still relies on a centralized master resulting in limited scalability. Similarly, [20] introduces a synchronization architecture, called the Synchronization State Buffer (SSB), which is a small buffer attached to the memory controller of each memory bank. It records and manages the states of active synchronized data units to support and accelerate word-level fine-grain synchronization. The SSB solution has been implemented in the context of the 160-core IBM Cyclops-64 chip architecture. The authors of [25] propose a novel thread synchronization technique relying on packet-switching mechanisms in a NoC-based manycore system. The interconnection system provides two physical links used in the protocol in order to avoid deadlocks. The work introduces a synchronization-operation buffer (SB), which enqueues and manages the requests issued by the processors. The mechanism uses a spin lock implementation, requiring a constant number of network transactions and memory accesses per lock acquisition. Note that the approaches described above do not support concurrent barriers if different application kernels are running in separate sub-regions of the manycore system.

3 Architecture

The proposed synchronization mechanism has been integrated in the GPU-like accelerator core being developed within the MANGO H2020 project. The core comes with a NoC-based manycore architecture, as shown in Figure 1, where each tile

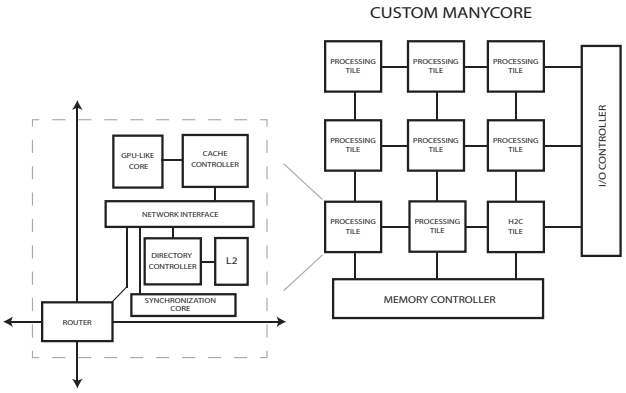

Fig. 1 The custom manycore system addressed by this work.

includes a hardware multi-threaded SIMD processor. The system provides a distributed coherent L2 cache and hardware coherence support based on a directory protocol. The main idea behind our solution is to provide a distributed approach inspired by the directory-based system. Our architecture aims to eliminate central synchronization masters, resulting in a better message balancing across the network. Combining a distributed approach, hardware-level messages, and a fine-grain control, the solution supports multiple barriers simultaneously from different core subsets. Coherence does not affect the overall performance, since the system provides a dedicated virtual network exclusively used for synchronization. Below we describe the main components of the proposed synchronization hardware.

Boot Setup. The Boot Setup module is in charge of initializing every structure involved in a barrier synchronization. It sends the `Setup` messages in the initialization phase, then sets and updates the barrier counters in the involved Synchronization Core. When an application kernel requires a synchronization, it specifies the barrier ID and the number of cores involved. The Boot Setup module steps into the Service state where it gathers all this information and selects a synchronization master. Then, when the network is ready, the Boot Setup initializes the chosen Synchronization Core by sending suitable messages through the NoC, conveying the barrier ID and the number of cores involved. From this point onward, the designated Synchronization Core becomes the synchronization master of this barrier, and it will handle all the synchronization messages from the involved cores.

Synchronization Core. The Synchronization Core is the key component of our solution. This module acts as the synchronization master, but unlike other hardware synchronization architectures, it is distributed among all tiles in the manycore. As explained above, the Boot Setup module selects a specific Synchronization Core based on the barrier ID set by the user. In this way, the architecture spreads the synchronization messages all over the manycore. By using this approach, the synchronization master is no longer a network congestion point. Figure 2 shows a simplified view of the Synchronization Core. The module is made of three stages: the first stage

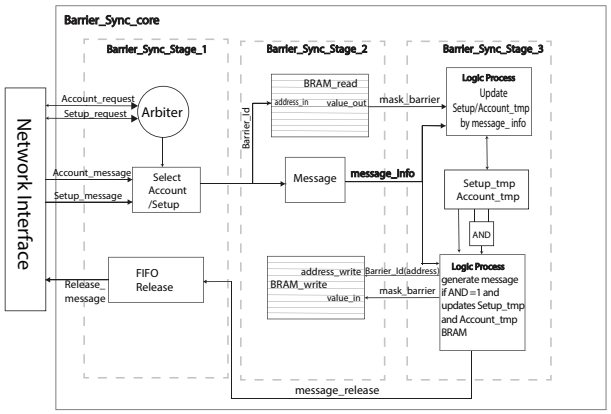

Fig. 2 Overview of the Synchronization Core.

selects and schedules the `Setup` and the `Account` requests. The selected request steps into the second stage, which strips the control information from the message. In the last stage, the stripped request is finally processed. If it is a `Setup` request, the counter is initialized with the number of involved threads. On the other hand, if the request is an `Account` message, the counter is decremented by 1. When the counter is 0, all the involved cores have hit the synchronization point, and the master sends a multicast `Release` message reaching all of them. Refer to Figure 3 for the format of the synchronization messages.

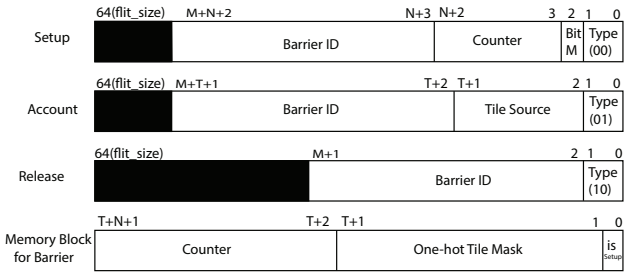

Fig. 3 Synchronization messages, 64 bits each. The `Setup` message is sent by the Boot Setup module in order to initialize the chosen synchronization master. It encloses the unique barrier ID and the number of threads to synchronize. The `Account` message is sent to the master, when a thread hits the barrier from the core side. The `Release` message is sent by the synchronization master to all the involved cores when all the `Account` messages are collected.

Barrier Core. The Barrier Core manages the synchronization on the core side. Each thread in a processing core can issue a barrier request through a specific barrier instruction introduced into the processor ISA. When a thread hits the synchronization point, the Barrier Core sends an `Account` message to the Synchronization Core, and stalls the requesting thread until a `Release` message from the master arrives.

An example of synchronization. Assume we have a manycore with 16 tiles: 14 processing tiles, one tile for the host interface, and one tile connected to the memory controller. The programmer executes a parallel kernel that involves the 14 processor tiles. As a first step, a core sends a `Setup` message to the chosen synchronization master, containing a counter value set to 14 and the unique barrier ID. The synchronization master is chosen by a simple module operation on ID: $T = ID \bmod TileNumber$. The Synchronization Core, located in tile T, receives those values and initializes a counter, that will be assigned to this specific barrier from this moment onward. The most significant bits of the barrier ID locate the tile that manages that ID, as in a distributed directory approach. On the core side, when a thread hits the synchronization point, the Barrier Core in its tile sends an `Account` message to the master in tile T and freezes the thread. As soon as the master receives this type of message, it decrements the respective counter. When the counter reaches 0, all the involved cores have hit the synchronization point, and the master sends to each of them a `Release` message by multicast. When receiving the `Release` message, the Barrier Core releases the thread and resumes the execution flow.

4 Implementation

In addition to the hardware, this activity also relied on the LLVM-based compiler developed within the MANGO project for the GPU-like core. We extended both the back-end and the front-end in the compiler in order to provide a C-level support to synchronization. On the back-end side, the processor ISA has been extended with ad-hoc instructions. On the front-end side, an intrinsic operation has been added, called `__builtin_nuplus_barrier_core(Id_Barrier, NULL)`. The synchronization components rely on a private virtual channel added in the system, which routes all synchronization messages. The baseline system already provided virtual channels but, as observed above, the hardware coherence support tends to flood the network infrastructure with data and coherence messages, and could easily impact the synchronization mechanism performance. On the core side, when a barrier instruction is decoded, the control unit waits until all previous instructions are committed, then it stalls the requesting thread, and fetches the barrier into the Barrier Core module in the execution stage (see Figure 4). This component sends an `Account` message to the designated Synchronization Core, and waits until a `Release` message arrives. We integrated the Boot Setup component in the Host Interface tile, which is normally used for communication and for booting the manycore system. An arbiter selects the proper component in the Host Interface tile depending on the commands coming in through the interface. The Synchronization Core has been integrated at the tile level, as shown in Figure 1. It is connected to the Network Interface of the processor tile and dispatches messages on the synchronization virtual channel. Below we will compare the performance of our distributed

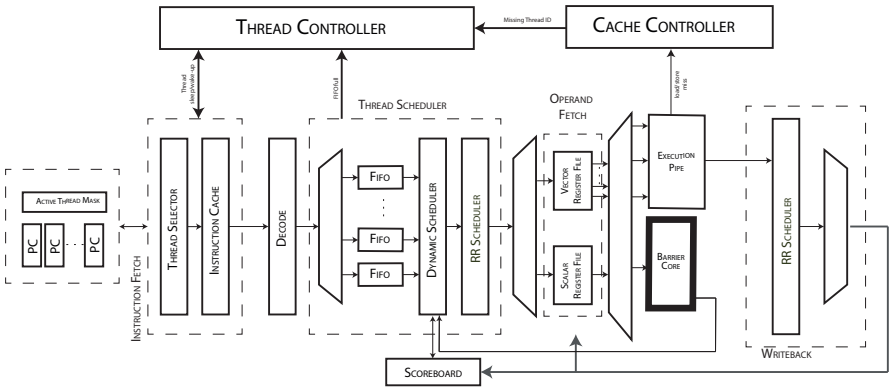

Fig. 4 The microarchitecture of the GPU-like core and the Barrier functional unit.

solution with a centralized master approach that allocates the Synchronization Core only in Tile 0.

5 Evaluation

In this section, we will evaluate the performance of the proposed solution in terms of clock cycles and area overhead. Next, we will compare the presented solution with a standard synchronization mechanism based on a centralized master. We will not compare our work with any software approach [25], since the proposed solution does not require memory accesses or atomic instructions, and thus incurs a significantly lower latency. A useful feature introduced by our solution is the support for multiple barrier synchronizations from different application kernels being executed simultaneously on different subsets of cores.

Simulation. For timing evaluation, we run the system with different NoC sizes by using a cycle-accurate simulator. The architecture has been successfully synthesized on a Xilinx xc7a100tcsg324-1 FPGA with two different versions of the synchronization master: a standard centralized configuration and our distributed approach. Table 1 summarizes the resource requirements of the two approaches for different NoC sizes. The areas occupied by the Barrier Core and the Boot Setup are constant, since those modules are not influenced by NoC parameters such as the number of tiles or specific topologies. The distributed approach incurs area costs increasing with the number of tiles, although it can support a larger number of barriers, which can involve selected sub-groups of threads, with a resource overead distributed uniformly across the tiles.

The proposed architecture and the centralized solution are simulated in the same environment with the same parameters. Both run the same kernel made of a barrier setup performed by Core 0, followed by a call to the synchronization intrinsic pro-

Table 1 Comparison of the resource requirements of the synchronization hardware

	Barrier Core	**Boot setup**	**Centralized Synchronization Core**			**Distributed Synchronization Core** (*per tile*)		
			2x2	4x2	4x4	2x2	4x2	4x4
LUT	69	16	257	324	572	185	207	268
Flip-Flop	56	17	369	520	786	321	420	567

vided by the compiler, as explained in Section 4. The time performance is evaluated by averaging out the synchronization times of all the involved threads. Each core has been equipped with a 64-bit performance counter which is initialized when it detects a barrier operation, and stopped when the release message from the master is received.

Synchronization of the whole manycore. As a first experiment, we run a kernel which involves all processing cores instantiated in the manycore. We compare the approach based on a centralized synchronization master with our distributed solution. Table 2 summarizes the clock cycles required by the synchronization for different NoC sizes. As observed, the two approaches reach the same results, due the limited size of the network and the absence of concurrent synchronizations.

Table 2 Time of a single synchronization operation involving all cores

NoC Size	Centralized	Distributed
2x2	26	26
4x2	72	72
4x4	74	74

Average clock cycles per thread

Synchronization with concurrent kernels. The proposed architecture supports multiple barrier synchronizations from different application kernels being executed simultaneously on different subsets of cores. The maximum number of distinct barriers supported is $N/2$, where N is the number of threads instantiated in the system. In the experiments below, we compare the number of clock cycles needed to synchronize all the subsets, running the maximum number of supported barriers in parallel. We assume that the developer parallelizes the kernel on sets of consecutive tiles.

Table 3 summarizes the results of our study. The centralized solution lacks scalability, even for small NoC configurations, and the final count is highly dependent on the position of the synchronization master. Our approach requires the same clock count for the smallest NoC, but results in improved scalability as the NoC size is in-

creased. Furthermore, our solution does not rely on a fixed master, as this is chosen based on the barrier ID, hence by the user or possibly by the compiler.

Table 3 Time of multiple independent synchronization operations taking place concurrently.

NoC Size	Centralized	Distributed
2x2	26	26
4x2	164	135
4x4	263	216

Average clock cycles per thread

6 Conclusions

This workshop paper presented the synchronization hardware added to the GPU-like custom manycore accelerator being developed within the MANGO H2020 project. The proposed solution relies on a distributed master and on a lightweight control unit on the core side, providing a synchronization mechanism through hardware-level message exchange without any memory access overhead. The proposed solution supports multiple barriers for different application kernels executed simultaneously on different subsets of cores. The results collected for different NoC sizes provided indications about the area overheads incurred by our solution and demonstrated the benefits of using a dedicated hardware synchronization support. As a long-term goal of this research, we aim to explore the implications of different NoC topologies as well as the impact of the positions of concurrent masters when multiple kernels are run in different subsets of cores.

Acknowledgments. This work is supported by the European Commission in the framework of the H2020-FETHPC-2014 project n. 671668 - MANGO: exploring Manycore Architectures for Next-GeneratiOn HPC systems.

References

1. K. Paranjape, S. Hebert, and B. Masson, “Heterogeneous computing in the cloud: Crunching big data and democratizing HPC access for the life sciences,” Intel, Tech. Rep., 2010.
2. M. Barbareschi, A. Mazzeo, A. Vespoli, “Network traffic analysis using Android on a hybrid computing architecture”, *Int’l Conf. on Algorithms and Architectures for Parallel Processing*, Springer, pp. 141–148, 2013.
3. T. Bjerregaard and S. Mahadevan. “A survey of research and practices of network-on-chip”, *ACM Computing Surveys*, vol. 38. no. 1, 2006.
4. A. Cilardo, E. Fusella, L. Gallo, A. Mazzeo, “Automated synthesis of FPGA-based heterogeneous interconnect topologies”, *Int’l Conf. on. Field Programmable Logic and Applications* (FPL), 2013.

5. F. Amato and F. Moscato, “Pattern-based orchestration and automatic verification of composite cloud services”, *Computers and Electrical Engineering*, vol. 56, pp. 842-853, 2016.
6. E. Fusella and A. Cilardo, “Lighting up on-chip communications with photonics: Design tradeoffs for optical NoC architectures”, *IEEE Circuits and Systems Magazine*, vol. 16, no. 3, pp. 4-14, 2016.
7. T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm, “A Survey of Barrier Algorithms for Coarse Grained Supercomputers”, *Chemnitzer Informatik Berichte*, 2004.
8. E. Fusella, A. Cilardo, “H2ONoC: A Hybrid OpticalElectronic NoC Based on Hybrid Topology”, *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 1, pp. 330-343, 2017.
9. M. Barbareschi, E. Battista, N. Mazzocca, S. Venkatesan, “A hardware accelerator for data classification within the sensing infrastructure”, *Int’l Conf. on Information Reuse and Integration* (IRI), IEEE, pp. 400–405, 2014.
10. F. Amato and F. Moscato, “Exploiting Cloud and Workflow Patterns for the Analysis of Composite Cloud Services”, *Future Generation Computer Systems*, vol. 67, pp. 255-265, 2017.
11. D. Wentzlaff, et al., “On-Chip Interconnect Architecture Of The TILE Processor”, *IEEE Micro*, vol. 27, no. 5, 2007.
12. S. Vangal, et al. “An 80-tile 1.28 TFLOPS network-on-chip in 65nm CMOS”, *IEEE International Conference on Solid-State Circuits* (ISSCC), IEEE, 2007.
13. G. Tan, V. C. Sreedhar, and G. R. Gao. “Analysis and performance results of computing betweenness centrality on IBM Cyclops64”, *The Journal of Supercomputing*, vol. 56, no. 1, pp. 1-24, 2011.
14. A. Olofsson, “Epiphany-V: a 1024 processor 64-bit RISC system-on-chip”, arXiv preprint arXiv:1610.01832, 2016.
15. I. Kuon and J. Rose, “Measuring the gap between FPGAs and ASICs”, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203-215, 2007.
16. A. Cilardo, E. Fusella, L. Gallo, A. Mazzeo, “Joint communication scheduling and interconnect synthesis for FPGA-based manycore systems”, *Design, Automation and Test in Europe Conference and Exhibition* (DATE), 2014.
17. J. R. Goodman, M. K. Vernon, and P. J. Woest, “Efficient synchronization primitives for large-scale cache-coherent multiprocessors”, *SIGARCH Comput. Archit. News*, ACM, vol. 17, no. 2, pp. 64-75, 1989.
18. O. Villa, G. Palermo, and C. Silvano, “Efficiency and scalability of barrier synchronization on NoC based manycore architectures”, *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pp. 81-90, 2008.
19. A. Cilardo, E. Fusella, L. Gallo, A. Mazzeo, “Exploiting concurrency for the automated synthesis of MPSoC interconnects”, *ACM Trans. on Embedded Computing Systems* (TECS) vol. 14, no. 3, pp. 57, 2015.
20. W. Zhu, et al. “Synchronization state buffer: supporting efficient fine-grain synchronization on manycore architectures”, *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 35-45, 2007.
21. Y.-L. Tseng, K.-H. Huang, and B.-C. C. Lai, “Scalable mutli-layer barrier synchronization on NoC”, *International Symposium on VLSI Design, Automation and Test* (VLSI-DAT), IEEE, 2016.
22. D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture, a Hardware/Software Approch*, Morgan Kaufmann, 1998.
23. A. Cilardo, E. Fusella, “Design automation for application-specific on-chip interconnects: A survey”, *Integration, the VLSI Journal*, vol. 52, pp. 102-121, 2016.
24. J. L. Abellán, J. Fernández, and M. E. Acacio, “Efficient Hardware Barrier Synchronization in manycore CMPs”, *IEEE Trans. on Parallel and Distributed Systems*, vol. 23, no. 8, pp. 1453-1466, 2012.
25. M. Monchiero, G. Palermo, C. Silvano, and O. Villa, “Efficient Synchronization for Embedded On-Chip Multiprocessors”, *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 10, pp. 1049-1062, 2006.