Design of a Collective Communication Infrastructure for Barrier Synchronization in Cluster-Based Nanoscale MPSoCs

José L. Abellán
Juan Fernández
Manuel E. Acacio
DiTEC, University of Murcia
30100 Murcia, Spain

Davide Bertozzi
ENDIF, University of Ferrara
44100 Ferrara, Italy

Daniele Bortolotti
Andrea Marongiu
Luca Benini
DEIS, University of Bologna
40136 Bologna, Italy

Abstract—**Barrier synchronization is a key programming primitive for shared memory embedded MPSoCs. As the core count increases, software implementations cannot provide the needed performance and scalability, thus making hardware acceleration critical. In this paper we describe an interconnect extension implemented with standard cells and with a mainstream industrial toolflow. We show that the area overhead is marginal with respect to the performance improvements of the resulting hardware-accelerated barriers. We integrate our HW barrier into the OpenMP programming model and discuss synchronization efficiency compared with traditional software implementations.**

I. Introduction and Related Work

Barrier synchronization becomes increasingly challenging as the level of integration in multi-processor systems-on-chip (MPSoC) keeps growing. There is today little doubt on the fact that software implementations are not suitable to provide the needed scalability of barrier synchronization in embedded systems and that some form of hardware support is essential.

Barrier optimization techniques in the embedded MPSoC domain often consist of optimized memory controllers or communication controller interfaces [3], which aim at reducing the overhead of busy wait synchronization algorithms. These approaches focus on accelerating the barrier logic (i.e., loop over the participants for gathering and releasing them) and removing memory and interconnect congestion by providing dedicated local polling registers. However, the exchange of synchronization messages takes place through the main system interconnect, typically a Network-on-Chip (NoC) [4]. This solution is however non optimal, since communication requirements for synchronization and large-grain data movements are very different, and thus it is difficult to devise a topology which efficiently satisfies both. Moreover, the mutual interference between the two traffic flows has to important negative implications to performance: first, it degrades QoS of applications; second, it fails to provide ultra-low latency synchronizations.

A few solutions for embedded MPSoCs propose the adoption of dedicated communication infrastructures to carry synchronization-related traffic [15], [14]. Current proposals [9], [1] advocate relying on non-standard implementation technology in order to keep up with scalability and efficiency as the number of cores increases in MPSoCs. In this paper, we rather make use of standard technology to explore different lightweight network infrastructures that are based on very simple connectivity patterns and extremely fast synchronization protocols. Nonetheless, efficiency and scalability are non-trivial to materialize when using standard technology because of several challenges. First, the RC propagation delay of on-chip interconnects degrades as feature sizes shrink, hence making global wires increasingly slow [7], [8]. Second, propagation delay of logic controllers required by each scheme affects their operating speed, again making relative performance non-trivial.

The most advanced MPSoC platforms achieve scalability through IP core clusterization and cluster replication [10], [11], where each cluster can potentially operate at an independent frequency for the sake of power efficiency. This architectural template is considered in this paper and adds two new variables to the design space. First, the most efficient hardware barrier implementation at the cluster-level may not be the same for the top level, where global synchronization between large clusters must be achieved. Second, conveying synchronization messages at the inter-cluster level is a globally asynchronous locally synchronous (GALS) communication issue that has never been adequately investigated before.

Our first contribution with this paper consists of a physical design space exploration of collective communication structures at the intra- and inter-cluster level in an advanced 45nm technology library. We investigate the most lightweight and performance-efficient connectivity pattern for different clustering granularities and layout sizes. As a second contribution, layout-aware performance of the most promising communication structures is annotated in the HDL (SystemC) models of a real-life MPSoC system. This virtual platform is enriched with a software stack composed of a OpenMP-based programming model, compiler and runtime system [12].

While our results show that in absolute terms a global, system-wide barrier always provides the smallest synchronization latency, the hierarchical barrier enables an interesting feature, namely it supports multiple co-existing HW barriers (one per cluster). This is a very important feature for large systems capable of running different applications which need to be synchronized independently.

II. *Hardware Support for Barrier Synchronizations*

To develop our barrier infrastructures we rely on mainstream industrial toolflows. Therefore, connectivity patterns are fully exposed to the effects of interconnect-dominated nanoscale technologies. To alleviate such an issue, our designs have been devised employing simple patterns and very low-bandwidth

978-3-9810801-8-6/DATE12/ © 2012 EDAA

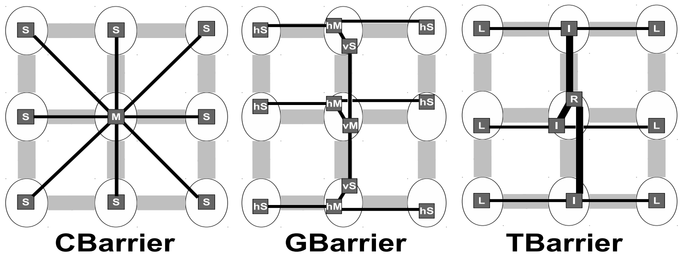

Fig. 1: Intra-cluster Barriers for a 9-core Cluster.

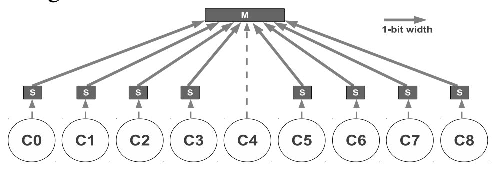

Fig. 2: Gather phase for *CBarrier* architecture.

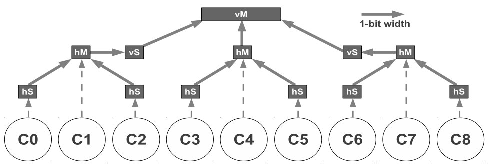

Fig. 3: Gather phase for *GBarrier* architecture.

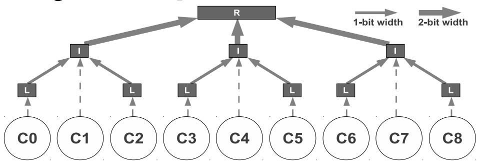

Fig. 4: Gather phase for *TBarrier* architecture.

on-chip *Links* (1-bit width in most cases). These considerations led us to discard the butterfly and the all-to-all barriers presented in [4]. Their high number of links make them unsuitable for a hardware implementation, especially in light of the high link inference cost (see Section III). We selected three barrier protocols better suited for silicon implementation: the *Central* barrier, the *GLine* barrier and the *Tree* barrier, illustrated in the following sections. Since the focus of this work is on clusterized systems, the hardware-barriers under test are explored within a single cluster, which we assume to be covered by a single clock domain, and among clusters, where clock domain crossing becomes an issue.

A. Intra-Cluster barriers

All the proposed intra-cluster barriers are based on the same two-phase protocol of a typical *Master-Slave* Barrier [4]: the *gather* phase and the *release* phase. During the *gather* phase a *master* thread waits for all the slaves to join the barrier. The latter notify their presence on some status flags, then wait for the master to free them during the *release* phase. To illustrate our hardware barriers, we consider an example of cluster architecture composed of 9 cores interconnected by a 2D-mesh NoC topology.

1) The Central Barrier architecture: The *CBarrier* is schematically shown in Figure 1. *Links* are represented with fine black lines while *controllers* are depicted as dark grey boxes. There are two kind of controllers acting as Master and Slave (M and S boxes, respectively). *CBarrier* features the minimum number of synchronization stages, a desirable property for hardware acceleration. The synchronization protocol for *CBarrier* relies on the exchange of 1-bit messages (signals) between master and slave controllers. Figure 2 represents the *gather* phase when all cores reach the barrier the same time. The release phase is exactly the same except for the notifications flowing in the opposite direction. Communication between controllers is depicted with solid lines.

2) The Gline-based Barrier architecture: GBarriers, shown in Figure 1, match the 2D mesh structure of the regular array fabric of PEs [1]. There are a number of *links* interconnecting four sorts of *Controllers*: horizontal (h) and vertical (v) master (m) and slave (s) controllers . Rather than leveraging full-custom *G-lines* technology and *S-CSMA* technique, in this paper we implement *GBarriers* with a standard cell design methodology. First, *G-lines* are implemented through conventional on-chip wires, allocating a different line per slave controller (in contrast to [1], where *G-lines* could be shared by different slaves connected to the same master controller). Second, we mimic the *S-CSMA* technique, that allows a master controller to determine the number of simultaneous slaves' signals transmitted over a particular *G-line*, by instructing the master to sample its different slaves' lines in a loop until all expected signals have been received. The synchronization protocol for *GBarrier* is depicted in Figure 3 for the *gather phase*. The *release phase* is exactly the same but the notifications flow in the opposite direction.

3) The Tree Barrier architecture: (*TBarrier*), shown in Figure 1, provides the best theoretical scalability, due to the lowest number of messages exchanged between master and slaves. Our implementation leverages three kind of *Controllers*: leaf nodes (*L*), internal nodes (*I*) and root nodes (*R*). The synchronization protocol for the *gather phase* is illustrated in Figure 4. The *R* controller is responsible for counting the number of participants. For the first phase, all *L* controllers send one 1-bit message to their corresponding *I* controller. When *I* has gathered all expected threads it notifies the *R* controller. Finally, *R* awaits messages from all *I* controllers.

Note that the *Link* between *I* and *R* controllers is wider than that between *L* and *I* controllers. In general, these *links* will be $log_2(Leaves)$-bit wide (e.g. 2-bit for the example in Figure) to carry the necessary messages for the *gather* phase. During the *release* phase only 1-bit width *links* are needed to notify the completion of the synchronization.

B. Inter-cluster Barriers

The targeted cluster-based MPSoC supports different clock speed for various clusters (see for example STM p2012 [11]). The main difference between global (inter-cluster) and local (intra-cluster) synchronization is that global controllers may communicate through different clock domains. Thus, we have designed inter-cluster barriers using asynchronous global Links (*gLinks*). To avoid metastability at the controller in the receiving cluster we use brute-force synchronizers. We found this kind of synchronization interface more suitable with respect to alternative ones such as dual-clock FIFOs because of the tiny width of our links and of the one-shot nature of communications over them. Also plausible clocking is not suitable due to the multi-port nature of controllers. Figure 5 shows our *CBarrier* architecture adapted to inter-cluster synchronization. For communication between master and slave controllers, we use two brute-force synchronizers (*BFsynch0* and *BFsynch1*).

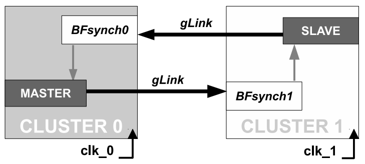

Fig. 5: Inter-cluster *CBarrier* architecture for 2 clusters.

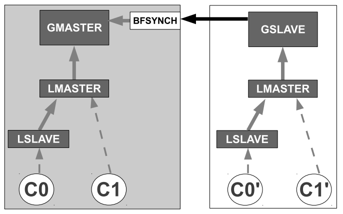

Fig. 6: Gather phase for *CBarrier* at intra- and inter-cluster levels.

Similar schemes for *GBarrier* and *TBarrier* can be implemented by simply adding one brute-force synchronizer for every *gLink*. It is worth noting that, every *BFsynch* is 1-bit width except for the *TBarrier* implementation. As pointed out in Section II-A3, communications between internal and root controllers utilize up to $log_2(Leaves)$ bits, thus requiring $log_2(Leaves)$-bit width *BFsynchs*. Figure 6 shows the complete synchronization process considering a platform composed of two 2-core clusters covered by two different clock domains. We adopt the *CBarrier* design for both inter and intra-cluster levels. As we can see, we distinguish between local and global controllers (L and G prefixes, respectively) depending on the level of the synchronization. Notice that, Local Master controllers have also been extended to locally communicate with their corresponding global controller enabling the interplay between the two levels. We highlight in black the arrow for the *gLink* among the two clusters. Using *GBarriers* and *TBarriers* for inter-cluster synchronization can be done in a similar way. Also, it is possible to use different protocols at each level of the design hierarchy.

III. Evaluation

As previously explained, to implement our synchronization support we use of a mainstream industrial synthesis toolflow and an STMicroelectronics 45nm standard cell technology library. Placement-aware logic synthesis is performed through Synopsys Physical Compiler. The final place-and-route step is performed with Cadence SoC Encounter which also involves clock tree synthesis. We assume a single clock domain with a unique clock tree for every cluster of the configurations. Finally, a sign-off procedure is run by Synopsys PrimeTime to accurately validate the timing properties of our designs.

Our techniques have been designed and assessed by defining non-routable obstructions. Such obstructions are placed to mimic the area of every core of the simulated systems. In this work, we assume that this area is equal to $0.55\times0.55\text{mm}^2$. Additionally, fences are defined to limit the area where the cells of each barrier's controller can be placed. Such obstructions and fences also ensure minimum-length routing for the links in order to reduce their impact on performance and area overhead as the wire length increases.

1) Intra-cluster Barriers: Each barrier's controller has been implemented by separating the delay that signals take along the wires from the effective computation that the controllers require to generate their output signals. For small clusters, the critical path is defined by the most complex barrier's controller (e.g. the Master for CBarrier), but as the wire length increases for bigger clusters, the wires could represent such critical path. Therefore, separating wire delays from controllers delays becomes essential in order to achieve maximum clock speeds. As a consequence, if all cores arrive at the barrier at the same time, the theoretical numbers of clock cycles that our intra-cluster barriers take are the following: 6 cycles for CBarrier; 14 for GBarrier; and, 10 for TBarrier.

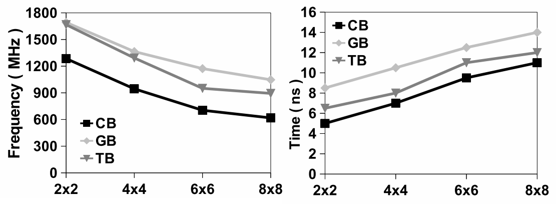

Fig. 7: Optimal frequencies and latencies for intra-cluster barriers.

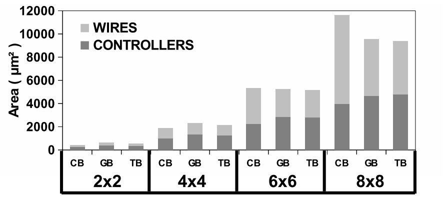

Fig. 8: Area overhead for intra-cluster barriers running at 600MHz.

Figure 7 depicts the maximum frequencies allowed by each intra-cluster barrier depending on the cluster size. As we can observe, each design can run faster depending on the number of steps required to perform a barrier synchronization. That is, the greater number of steps, the higher the frequency that can be achieved. For this reason, the highest frequencies are obtained for the GBarrier design. Moreover, as the size of the clusters becomes larger, the timing of critical paths obtained for barrier controllers and wires will be longer, what translates into lower achievable frequencies. Regarding barrier latencies, CBarrier completes in the smallest number of cycles for all configurations, despite its lower frequencies.

Figure 8 illustrates the area overhead for our intra-cluster barriers synthesized at 600MHz. The area devoted to wires constitutes the dominating factor for the three mechanisms. Therefore, since CBarrier has the longer links, it shows the highest overhead, which worsens as the cluster grows up. For the maximum performance settings (highest frequencies) faster and greater cells (higher drive strengths) are instantiated. This reduces the area gaps illustrated in Figure 8, but leads to identical conclusions.

2) Inter-cluster Barriers: To evaluate our inter-cluster barriers, we consider two different platforms composed of 2x2 and 4x4 clusters, assuming 4x4 cores per cluster as in [11]. At this level of synchronization, *gLinks* could introduce an unpredictable delay since they are considered as asynchronous and potentially unconstrained by the routing tool. Since we are targeting ultra-low latency communications, we explicitly constrain propagation delay across asynchronous links by setting a *set_max_delay* command to a quasi-zero value for every *gLink* in the three designs. As explained above in Figure 6, the synchronization protocol for a system composed of multiple clusters/clock domains is split into two levels of synchronization. The first level, that implements a single intra-cluster barrier for every cluster/clock domain and the top level, that is implemented by employing inter-cluster

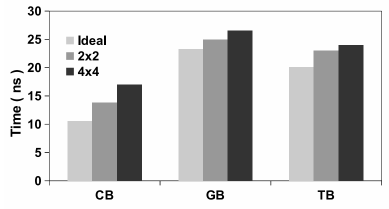

Fig. 9: Latency for inter-cluster barriers running at maximum frequency.

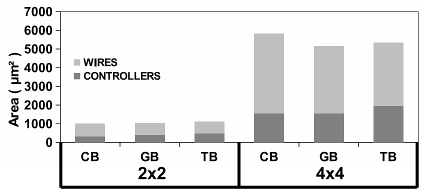

Fig. 10: Area overhead for inter-cluster barriers.

barriers crossing different cluster/clock domains. Therefore, the maximum frequency at top level for a particular inter-cluster controller is limited by the maximum operating speed achieved at the first level and vice-versa. For the scenarios discussed above, we obtained that the maximum operating speed is imposed by the intra-cluster level. All of the barrier controllers have been synthesized at 950MHz (the maximum frequency in Figure 7 for a 4x4-core cluster).

Figure 9 shows the barrier latencies for 2x2 and 4x4 clusters running at max speed. These latencies correspond to the inter-cluster barrier operation without adding up intra-cluster time. We also depict the barrier delay that the three designs take in theory running at the target frequency. As we can observe, the most efficient implementation is still CBarrier for these particular configurations. The only case when CBarrier is not the best option is when *gLinks* are too long, which nullifies the benefits of having a smaller number of stages. As expected, the length-delay phenomenon of *gLinks* is more pronounced for the CBarrier architecture, as compared to theoretical barrier timings. Nonetheless, this is not enough for outperforming both the GBarrier and TBarrier designs. We analyze in depth this issue by using a sign-off tool (i.e. Synopsys PrimeTime). This tool reports that the timing of links as a function of their lengths are as follows: 0.7, 1.2 and 2.2 ns; for 2.2, 4.4 and 8.8 mm respectively. As our architectures do not use longer links than 8.8 mm, negligible penalties in latency are reported what explains a higher efficiency for the inter-cluster CBarrier.

Figure 10 shows the area overhead for the three designs (not including intra-cluster). As we can see, *gLinks* is the most demanding part of the designs. CBarrier has thus the highest area overhead due to its longest links.

Finally, since every cluster could operate using a different clock domain, we have also analyzed this issue for our inter-cluster barriers. In particular, we have used two frequencies

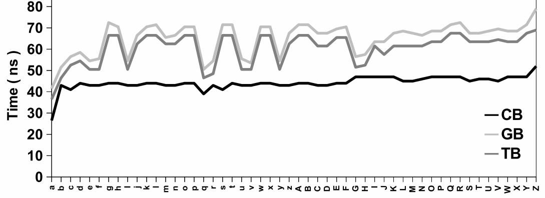

Fig. 11: Inter-cluster latencies using a combination of 300 and 600MHz.

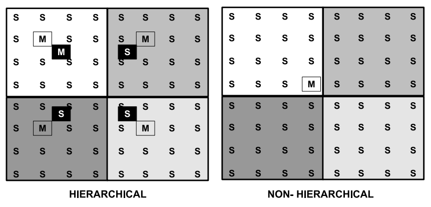

Fig. 12: Hierarchical and Non-Hierarchical (Flat) CBarrier architecture.

TABLE I: Performance statistics for CBarrier designs.

	Frequency	Latency	Area Wires	Area Controllers
Hierarchical	950MHz	22ns	4,935μm^2	5,922μm^2
Flat	620MHz	17ns	6,137μm^2	7,977μm^2

(300 and 600 MHz) that have been assigned throughout the 16 clusters in a combinatorial way (see x-axis in Figure 11). As we can observe, inter-CBarrier is the most efficient implementation and barrier delay increases from left (all clusters at 600MHz) to right (all clusters at 300MHz), that is, from the fastest to the slowest configuration.

3) Clusterization overhead: So far, the cluster structure of the MPSoC system has been equally reflected in the inter and intra-cluster barrier schemes. Such an approach has an unique advantage: it is capable of supporting multiple co-existing barriers in the system. Indeed, each of the controllers can be independently programmed by the software, thus enabling disjoint synchronization domains. In this section, we aim at quantifying the overhead with respect to a flat interconnect solution. As a case study, we consider a 64-core platform split into four different clusters/clock domains in which independent applications could run simultaneously on the available 16 cores (the maximum cluster size employed in [11]). From Sections III-1 and III-2, we derive that the inter- and intra-CBarrier architectures are the preferred choice for the target platform when the hierarchical approach is taken (see Figure 12). Black boxes represent the top level of the hierarchy (inter-cluster controllers). We compare the hierarchical CBarrier with a flat layout composed of a single level in which all controllers are logically placed and connected through the CBarrier pattern to a centralized global master (see flat scheme in Figure 12).

In Table I, we report the performance results in terms of maximum frequency, barrier latency and area overhead for the hierarchical and flat CBarrier layouts. As we can see, the maximum frequency is achieved by the hierarchical design. This is due to the fact that for both scenarios the Master controllers constitute the critical path to performance. Then, the more complex the Master, the lower frequency will be achieved. That is the reason why since there is only one Master in the flat design, this can support a lower operating speed (see the Table). Also regarding latency the flat solution is the most efficient one. Indeed, the hierarchical layout nearly doubles the number of steps (clock cycles) required by the flat design. Regarding area overhead, the flat scheme is the most consuming design. This it is mainly due to the very high number of brute-force synchronizers that this layout requires. Moreover, the average link length of the flat design is longer than the hierarchical layout.

IV. Full-system Simulation

As a final exploration, we want to assess the impact of coupling our hardware barrier proposals (HW barriers) with a real-life software stack. To this aim, we developed SystemC models of the two main components of our hierarchical barriers, namely the local and global controllers described in Section II. We annotated these models with the latencies extracted from the characterization presented in Table I. The models are finally integrated in a cycle-accurate full-system simulator which allows us to build an instance of the 64-core, 4-cluster MPSoC considered in Section III-3. Clusters are interconnected through a global NoC. Each of them features 16 cores, communicating through a fast multi-banked, multi-ported Tightly-Coupled Data Memory (TCDM). The number of memory ports in the TCDM is equal to the number of banks to allow concurrent accesses to different banks. Conflict-free TCDM access have two-cycles latency.

To accurately account for the overheads introduced by a realistic software stack, we integrate our HW barriers into a widespread programming model such as OpenMP. In OpenMP, parallelism is specified at a very high level by inserting directives (annotation) to a sequential C program. The compiler is responsible for translating these directives into parallel threads of execution. Many of the parallelization services provided by OpenMP are implemented within a runtime library, queried by the parallel threads. We re-wrote the low level OpenMP runtime primitives for barrier synchronization so that we can select between an optimized software implementation and the invocation of our HW controllers.

As a software implementation, we consider the topology-aware variant of the *tree* barrier discussed in [13]. We choose this barrier because it is reported in literature as one of the best-performing for distributed systems and because it has in practice an analogous behavior to the hardware counterpart explored in this paper. In the SW implementation, one single core in the system acts as a *global master*, one core per cluster acts as a *local master*, while the rest of the cores act as a *local slave* (see hierarchical scheme in Figure 12).

The synchronization process for our SW barrier is as follows. First, in the *Local Gather* phase, each of the local masters wait for each of its slaves to notify its arrival at the barrier on a private status flag (LOCAL_NOTIFY array). After arrival notification, local slaves check for barrier termination on a separate private location (LOCAL_RELEASE array). Second, in the *Global Gather* phase, the global master waits for all local masters to notify their arrival in a private status flag (GLOBAL_NOTIFY array). After arrival notification, local masters wait for global synchronization termination on a local flag (GLOBAL_RELEASE). Third, in the *Global Release* phase, the global master notifies the termination of the global synchronization step by writing into each local master's GLOBAL_RELEASE flag. Fourth, in the *Local Release* phase, each local master notifies the termination of the whole barrier by writing into each local slave's private flag from the LOCAL_RELEASE array. Moreover, to prevent polling activity from injecting interfering traffic on the interconnect, we distribute notification and release flags so that each processor does busy waiting on a private, local memory cell. In particular, we leverage the multi-banking feature of on-cluster TCDMs to make sure that each slave directs its polling transactions to a different memory bank.

Regarding the HW barriers, we considered both the *Hierarchical* and *Flat* barriers described in Section III-3. The number of threads involved in a parallel region can be set by the programmer with the clause `num_threads` when invoking the OpenMP `#pragma omp parallel` directive. Our barriers are capable of synchronizing a smaller number of cores than the total, but a *Setup* phase is necessary to appropriately program the controllers. For the *Flat* HW barrier this setup only consists of a write to a controller's `max_events` memory-mapped register. For the *Hierarchical* HW barrier and for the SW barrier the setup is slightly more complex. Based on the number of cores in each cluster (`CPC`) and the number of threads participating in the parallel region, it is necessary to figure out the number of clusters involved in the synchronization operation (`FC`). This number must be annotated into the memory-mapped register of the global controller. All the processors belonging to the first $\texttt{FC}-1$ clusters will take part to the barrier, thus corresponding local master controllers must be programmed to synchronize all of them. On the contrary, not all the processors belonging to the `FC`-*th* cluster may be involved in the barrier, thus the appropriate number must be computed and registered in the pertinent local master controller.

The more natural way to integrate barrier setup into the OpenMP execution model is to let the master thread accomplish this programming stage upon parallel region creation. We thus inserted this set of write operations inside the runtime library function `parallel_start`. These setup operations are regular writes into memory mapped registers. The corresponding transactions travel through the NoC, directed to each active cluster's controller. Once every controller has been programmed, the underlying hardware mechanism can be triggered via SW by writing/reading into two registers that every controller integrates: `bar_reg_in`, that is used when a core wants to participate in the barrier (to begin the gather phase) and `bar_reg_out`, that the controller writes to wake up its core to resume execution (the completion of the release phase).

As a first experiment we compare the cost, in terms of cycles, of both HW and SW barrier implementations. To avoid measuring wait time due to misaligned thread arrival on the SW barrier, we measured SW barrier time from the master thread, ensuring that all slaves have already entered the barrier. The barrier cost breakdown for the SW tree barrier is shown in Figure 13. The SW barrier costs approximately 700 cycles, considering the net time for gathering and releasing slaves locally and globally. An additional hundred cycles are induced by call overheads in the OpenMP runtime environment, plus the cost to initialize the barrier itself (setup phase), which amounts to 104 cycles. Overall, synchronizing 64 cores from OpenMP costs slightly more than 900 cycles. In Figure 14 we report the cost for the two HW barrier implementations. It is possible to see that, while the barrier time itself is not very different in the two cases, the setup phase, as expected, takes quite longer for the *Hierarchical* barrier. It has however to be pointed out that the cost for the setup phase has to be paid only when opening a parallel region with a different number than the previous. If the number of threads does not change among two parallel regions, the cost for the setup phase is drastically reduced (around 15 cycles).

As a second experiment, we want to estimate the granularity of parallelism enabled by the different barrier implementations. To this aim we use a small synthetic loop, which repeatedly invokes a small assembly routine composed uniquely of ALU instructions (to avoid memory contention effects). This routine is annotated with a `#pragma omp parallel` directive, which replicates its execution among all

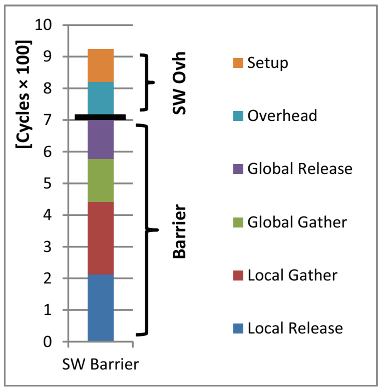

Fig. 13: SW barrier cost

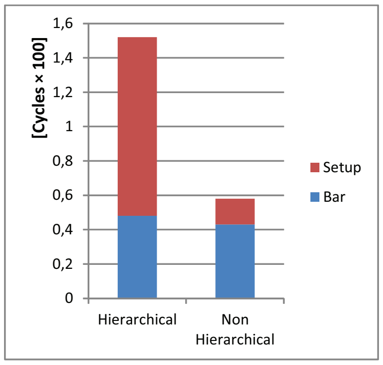

Fig. 14: HW barriers cost

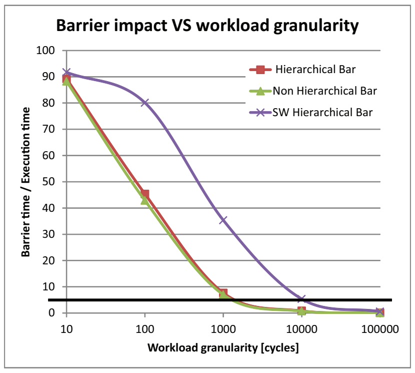

Fig. 15: Barrier cost / Workload

the participating threads. This routine can be parameterized to generate increasing granularities of parallel tasks (10 to 10000 cycles), so as to study how the parallelism is affected by barrier time. Plots are shown in Figure 15 for both HW and SW barriers. Here we show the percentage of time spent on synchronization when the granularity of the parallel task (i.e. the cycles taken for its execution) increases. For extremely small tasks (10 cycles) the barrier time is dominating in all cases ($\geq$ 90%). However, it is possible to see that clearly HW barriers cut down on latency by one order of magnitude with respect to the SW barrier. If we qualitatively establish that a 5% overhead for synchronization is negligible, it is possible to see that this point is reached by HW barriers at a granularity of thousand cycles, while this same point is reached by the SW barrier at ten thousand cycles. It is also worth underlining that from the software perspective the difference in latency between the two HW barriers is not appreciable, since the overheads introduced by the software stack tend to hide it.

V. Conclusion

Designing a dedicated collective communication infrastructure for synchronization signaling with standard design tools and technology libraries is a challenging task since it is directly exposed to the effects of interconnect-dominated nanoscale technologies. In spite of this, hardware barriers with the lower number of stages proved the most performance efficient in our physical implementation framework although using longer links. This is because the interconnect delay is not such to offset the inherent lower cycle count these schemes take to synchronize the system. Where instead the interconnect delay plays a role is in determining area of the hardware barrier, since the place-and-route tool operates aggressive repeater insertion to sustain performance over long links. However, from the software perspective the picture changes slightly. When integrating our HW barriers into complete software stacks (i.e., a programming model and its runtime environment) we saw that the difference in latency between the most performance-efficient implementation and the second best is not that relevant, because the software support for parallelism creation introduces sources of overhead that tend to hide it. Our experiments with OpenMP demonstrate that both the explored HW barrier solutions enable one order of magnitude-finer grained parallelism than pure-software implementations. Moreover, the hierarchical barrier allows to independently synchronize multiple processor groups (one per cluster) concurrently. This is something that is extremely important, for example, when nested parallelism comes into play, or when multiple disjoint parallel applications are running on the system. We plan to further explore this opportunity as future work.

Acknowledgment

This work was supported by projects FP7 VIRTICAL(288574) and JTI SMECY (ARTEMIS-2009-1-100230), funded by the European Community. Moreover, this work was also supported by the Spanish MEC and MICINN, as well as European Commission FEDER funds, under Grants "CSD2006-00046" and "TIN2009-14475-C04". José L. Abellán is supported by fellowship 12461/FPI/09 from Fundación Séneca - Agencia de Ciencia y Tecnología de la Región de Murcia.

References

[1] J. L. Abellán et al. A G-line-based Network for Fast and Efficient Barrier Synchronization in Many-Core CMPs. In Proceedings of the 39th International Conference on Parallel Processing, 2010.
[2] J. Sartori and R. Kumar. Low-Overhead, High-Speed Multi-core Barrier Synchronization. In Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers, 2010.
[3] M.Monchiero ivate/publications/TPDS11/cameraReady al.. Efficient Synchronization for Embedded On-Chip Multiprocessors. In IEEE Transactions on Very Large Scale Integration Systems, vol.14, no.10, October 2006.
[4] O. Villa et al.. Efficiency and Scalability of Barrier Synchronization on NoC Based Many-core Architectures. In Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, 2008.
[5] T. Krishna et al.. Express Virtual Channels with Capacitively Driven Global Links. In Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, 2008.
[6] E. Bolotin et al., QNoC: ivate/publications/IPDPS11/submission architecture and design process for network on chip, In J. Syst. Archit., vol.50, issue 2-3, pp.105-128, 2004.
[7] D. Ludovici et al., Capturing Topology-Level Implications of Link Synthesis Techniques for Nanoscale Networks-on-Chip, In Proceedings of the 19th ACM Great Lakes symposium on VLSI, 2009.
[8] A. Pullini et al., Bringing NoCs to 65 nm, In IEEE Micro, vol.27, issue 5, pp.75-85, 2007.
[9] J. Oh et al., TLSync: Support for Multiple Fast Barriers Using On-Chip Transmission Lines, In SIGARCH Comput. Archit. News, vol.39, issue 3, pp.105-116, 2011.
[10] Plurality Ltd. The HyperCore Processor. www.plurality.com/hypercore.html.
[11] ST Microelectronics and CEA. Platform 2012: A Many-core programmable accelerator for Ultra-Efficient Embedded Computing in Nanometer Technology. www.2parma.eu/images/stories/p2012_whitepaper.pdf, 2010.
[12] www.openmp.org. OpenMP Application Program Interface v.3.0. www.openmp.org/mp-documents/spec30.pdf
[13] A. Marongiu et al. Supporting OpenMP on a multi-cluster embedded MPSoC. Microprocessors and Microsystems www.sciencedirect.com/science/article/pii/S0141933111001001
[14] J. Sartori and R. Kumar. Low-Overhead, High-Speed Multi-core Barrier Synchronization. In *Proceedings of 5th International Conference on High Performance Embedded Architectures and Compilers*, 2010.
[15] C. Cascaval et al. Evaluation of a Multithreaded Architecture for Cellular Computing. In *Proceedings of 8th International Symposium on High-Performance Computer Architecture*, 2002.
[16] B. Chapman et al. Implementing OpenMP on a High Performance Embedded Multicore MPSoC. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium, 2009.
[17] W.-C. Jeun and S. Ha. Effective OpenMP Implementation and Translation for Multiprocessor System-On-Chip Without Using OS. In Proceedings of the Asia and South Pacific Design Automation Conference, 2007.
[18] F. Liu and V. Chaudhary. A Practical OpenMP Compiler for System on Chips. In Proceedings of the OpenMP Applications and Tools 2003 International Conference on OpenMP Shared Memory Parallel Programming, 2003.