Handling Shared Variable Synchronization in Multi-core Network-on-Chips with Distributed Memory

Xiaowen Chen[†,‡], Zhonghai Lu[‡], Axel Jantsch[‡] and Shuming Chen[†]
[†]National University of Defense Technology, 410073, Changsha, China
[‡]KTH-Royal Institute of Technology, 16440 Kista, Stockholm, Sweden
[†]{xwchen,smchen}@nudt.edu.cn [‡]{xiaowenc,zhonghai,axel}@kth.se

***Abstract*—Parallelized shared variable applications running on multi-core Network-on-Chips (NoCs) require efficient support for synchronization, since communication is on the critical path of system performance and contended synchronization requests may cause large performance penalty. In this paper, we propose a dedicated hardware module for synchronization management. This module is called *Synchronization Handler (SH)*, integrated with each processor-memory node on the multi-core NoCs. It uses two physical buffers to concurrently process synchronization requests issued by the local processor and remote processors via the on-chip network. One salient feature is that the two physical buffers are dynamically allocated to form multiple virtual buffers (a virtual buffer is related to a shared synchronization variable) so as to improve the buffer utilization and alleviate the head-of-line blocking. Synthesis results suggest that the SH can run over 900 MHz in 130nm technology with small area overhead. To justify the SH-enhanced multicore NoCs, we employ synthetic workloads to evaluate synchronization cost and buffer utilization, and run synchronization-intensive applications to investigate speedup. The results show that our approach is viable.**

I. Introduction

NoC based multi-core systems are promising solutions to the modern and future processor design challenges. Exploiting efficient parallelism and hence achieving high performance of parallelized applications running on multi-core Network-on-Chips (NoCs) [1][2] require high bandwidth memory subsystem, as well as efficient synchronization mechanisms. Careful design of the synchronization should be carried on, meeting strict design constraints in terms of performance. In this paper, we focus on the architectural support for efficient synchronization, targeting multi-core NoCs. Our work is motivated by the three points below.

1) Multi-core NoCs still display significant communication overhead amongst processor nodes as traditional synchronization mechanisms are employed, such as *spin lock*, which are based on the *busy-wait* techniques to ensure mutual exclusion through continuous polling of a shared synchronization variable. These synchronization mechanisms generate large network traffic, resulting in heavy contention.
2) The software solution and the hardware solution are two alternatives addressing the synchronization problems. In software solution, various synchronization primitives are implemented and used based on the underlying hardware architecture, which provide certain atomic operations, such as the pair of *load-linked* and *store-conditional* instructions and an atomic *test-and-set* instruction. However, its generality often comes with high synchronization overhead. Therefore, for the sake of obtaining high efficient synchronization, the dedicated hardware solution is a good choice.
3) The architecture of multi-core NoCs features a packet-switched network and distributed computing resources and storage elements. "*Distribution*" is one of architectural characteristics of multi-core NoCs. In generality, centralized solution will lead to overwhelming traffic contention near the central synchronization handling node when multiple processor nodes compete for a synchronization variable. Therefore, distributed solution is a good alternative to fit the architectural feature of multi-core NoCs.

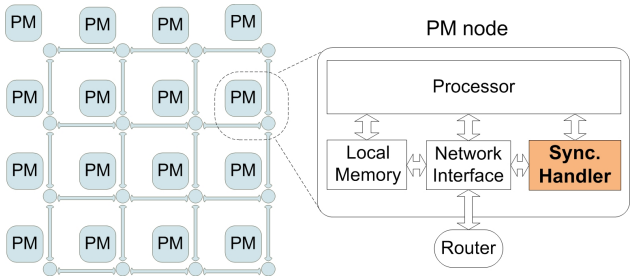

Fig. 1. (a) A 16-node mesh multi-core NoC, and (b) Processor-Memory node

Based on the aforementioned three motivations, we propose a dedicated and distributed synchronization hardware module, named *Synchronization Handler (SH)*, targeting multi-core NoCs. Fig. 1 (a) shows an example of our multi-core NoC architecture. The system is composed of 16 Processor-Memory (PM) nodes interconnected via a packet-switched network. The network topology is a mesh, which is a most popular NoC topology proposed today [3]. As shown in Fig. 1 (b), each PM node hosts a processor, a local memory, a Network Interface (NI) and a Synchronization Handler (SH). The SH module is connected to the processor and the Network Interface (NI). The SH offers a set of synchronization variables, which are globally addressed and accessed by all nodes. The synchronization variables are used as *locks* to provide mutual exclusion when shared memory references occur. Considering that, at each node, there are at most two requests coming simultaneously (one is from the local processor, the other from the network) in our multi-core NoC architecture, the SH features two physical buffers to serve synchronization requests issued by the local processor and remote processors via the on-chip network concurrently. The two physical buffers are dynamically allocated and logically organized as multiple virtual buffers to improve their utilization. This organization can avoid both consuming much area cost resulting from maintaining a physical buffer for each lock and one synchronization request blocking another independent synchronization request. To evaluate the SH's performance, we apply both detailed synthetic workloads and synchronization-intensive application workloads on our cycle-accurate multi-core NoC platform. The experimental result evaluates the cost of successful synchronization, the utilization of dynamically allocated buffers and the speedup of applications.

The rest of the paper is organized as follows. Section II discusses related work. Section III details the SH micro-architecture and its operation mechanism, and gives hardware synthesis results. Section IV reports simulation results with synthetic and application workloads. Finally we conclude in Section V.

II. Related Work

As a multiprocessor system, multi-core NoCs try to boost overall performance by exploiting efficient parallelism of applications running on it. Efficient synchronization support is important in exploiting efficient parallelism. Synchronization attracted a large body of research in multiprocessor systems. Recently, many researchers shifted to discuss synchronization issues in a single chip, including communication overhead [4][5][6][7][8], energy efficiency [4][9], scalability [7][9], etc.

Regarding efficient synchronization mechanism, in [5], Sampson presented a novel mechanism for barrier synchronization on chip multiprocessors (CMPs). His contribution is on barrier synchronization rather than lock/semaphore synchronization. In [8], Zhu proposed a scalable architectural design, which records and manages the states of frequently synchronized data, for fine-grain synchronization that efficiently performs synchronizations amongst concurrent threads. In [4][6], Monchier explored an optimization technique of synchronization mechanisms for shared memory MPSoCs based on NoC and targeted eliminating large contention incurred by re-spinning a lock. A synchronization hardware module was designed to augment the memory controller to support polling a lock locally in the shared memory. However, since all synchronization requests issued by all nodes flow through the network into the synchronization module, contention latency induced by heavy traffic near the module brings negative performance. In [9], Yu proposed a synchronization architecture for embedded multiprocessors to effectively implement the queued-lock semantics in a completely distributed way. In their multiprocessor platform, each processing node hosts a synchronization controller. However, 1) their multiprocessor platform uses the bus rather than complex interconnect (e.g. packet-switched network), and 2) the announcement of remote lock acquire request in the bus induces much traffic and is lack of scalability.

III. Synchronization Handler (SH)

In this section, we detail the SH architecture, its operation mechanism and the hardware cost.

A. Architecture

The SH consists of a Synchronization Variable Pool, a Scheduling Logic, two Physical Buffers and a Crossroad (see Fig. 2).

Synchronization Variable Pool:

it contains **N** locks with 1 bit each. There are two access ports for these locks. One is connected to Physical Buffer 1 and the other is connected to Physical Buffer 2. That is to say, it supports two concurrent accesses. All of locks in the pools of all nodes are globally addressed. All locks' status are back to the Scheduling Logic for controlling the Physical Buffers and guaranteeing the correctness of synchronization handling.

Physical Buffer:

Two physical buffers in the SH are adopted to permit responding to two simultaneous synchronization requests from both the local processor and remote processors via the on-chip network. Each physical buffer owns a buffer queue with the depth of **Q** and can receive synchronization requests from either the local processor or the network. The bypass path in the physical buffer allows that some requests are able to access the Synchronization Variable Pool directly without residing in the queue. The physical buffers are controlled by the Scheduling Logic to buffer incoming requests and to forward the request, which meets the synchronization requirement, to the pool. Moreover, the buffer is not a sequential queue. Logically, it's organized as several virtual buffers.

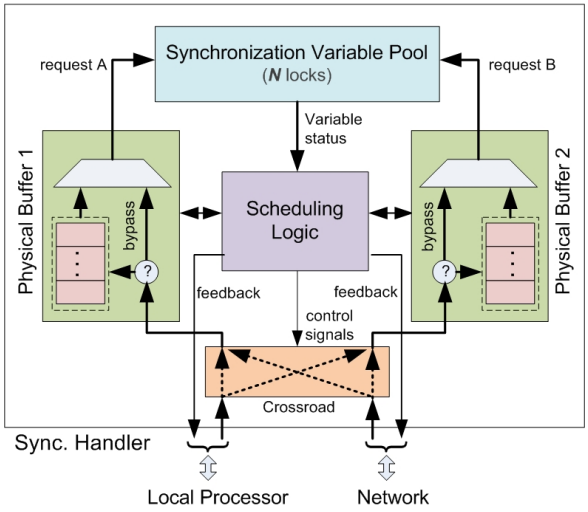

Fig. 2. The structure of the Synchronization Handler (SH)

The Multi-core NoC architecture characteristics lead to at most two requests coming simultaneously to the SH. The one is from the local processor and the other is from the remote processor via the network. Although processes or threads on the local processors could possibly send out their independent requests simultaneously, these requests are serialized by the local processor and sent to the SH one by one. It's the same for the remote requests which are serialized by the switches in the network and routed to the SH one by one. Therefore, two physical buffers are enough to meet the concurrent processing requirement.

To improve the two physical buffers' utilization, to avoid consuming much area cost resulting from maintaining a physical buffer for each lock, and to avoid one synchronization request blocking another independent synchronization request, the two physical buffers are dynamically allocated according to coming synchronization requests and logically organized as **N** virtual buffers (equals the number of locks). Each virtual buffer is coupled with a lock. Under this organization, synchronization requests accessing different locks go through their independent virtual buffers and do not interfere with one another. The buffer's dynamic allocation is maintained by the Scheduling Logic. Section III-B details how the virtual buffers work.

Crossroad:

The Crossroad receives control signals from the Scheduling Logic to automatically dispatch the synchronization requests to the proper physical buffers.

Scheduling Logic:

The Scheduling Logic acts as the central controller in the SH. It controls the Crossroad to determine directions of synchronization requests. It allocates the two physical buffers dynamically and organizes them to be **N** virtual buffers logically. It monitors all locks' status to maintain these virtual buffers and to perform correct actions on the coming synchronization requests. It acquires and releases the corresponding lock in an efficient way to reduce the contention and the overhead for improving the response time.

In summary, the SH (i) features a set of synchronization variables and two physical buffers, (ii) enables concurrent processing of synchronization requests from the local processor and remote processors via the on-chip network, and (iii) allocates the two physical buffers dynamically and organizes them into multiple virtual buffers to support efficient synchronization.

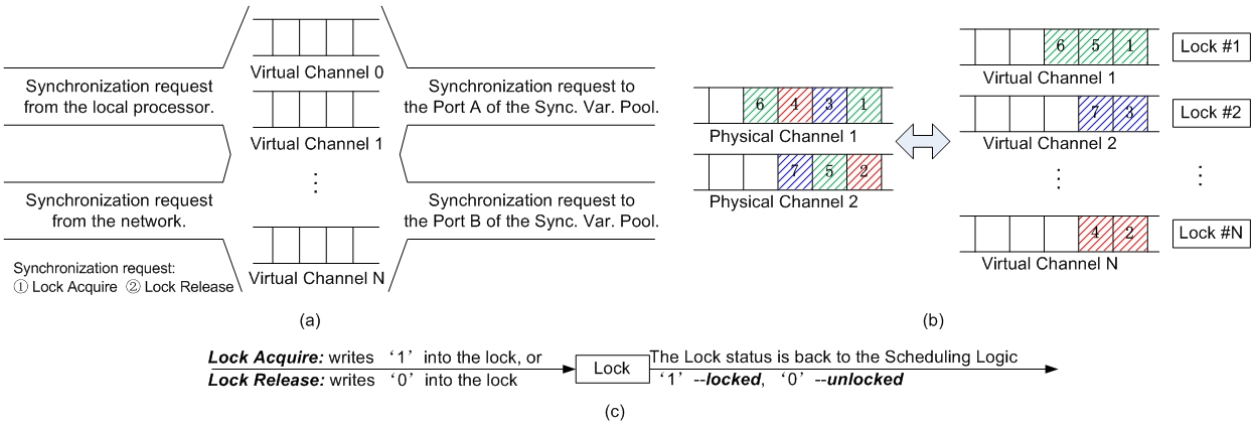

Fig. 3. (a) Buffer's dynamic allocation and "virtual buffers" organization, (b) an example describing how the two physical buffers are allocated dynamically and organized as multiple virtual buffers, and (c) two kinds of synchronization actions on locks

B. Operation Mechanism: Dynamic Buffer Allocation

The SH can respond to synchronization requests from the local processor and remote processors via the on-chip network concurrently. The two physical buffers are dynamically allocated and logically organized as multiple virtual buffers. As shown in Fig. 3 (a), synchronization requests come from the local processor and the network, go through the corresponding virtual buffers and finally enter the Synchronization Variable Pool. The requests are differentiated into two types: *Lock Acquire* and *Lock Release*. The "*Lock Release*" request always goes along the bypass path, while the "*Lock Acquire*" request also can go along the bypass path only when the related virtual buffer is empty. That is to say, the virtual buffers are only used for the "*Lock Acquire*" requests.

Fig. 3 (b) shows an example of how the two physical buffers are allocated dynamically and organized as multiple virtual buffers. Assuming that the Lock #1, #2 and #N are used, there are seven "*Lock Acquire*" requests coming from the local processor or remote processors. Request 1, 5 and 6 acquire Lock #1, Request 3 and 7 acquires Lock #2, and Request 2 and 4 acquires Lock #N. Because Lock #1, #2 and #N are used, the seven requests are buffered in the physical buffers. Their locations in the physical buffers reveal the order of their arrivals. In functionality, these requests are re-ordered according to the locks they acquire. Hence, Request 1, 5 and 6 logically form Virtual Buffer 1, Request 3 and 7 logically form Virtual Buffer 2, and Request 2 and 4 logically form Virtual Buffer N.

By monitoring all locks' status, the Scheduling Logic performs actions to guarantee the correctness and efficiency of locking. The lock status is categorized into two types: ***unlocked*** – the lock is not being used by any processors and ***locked*** – the lock is being used by a certain processor. Fig. 3 (c) describes synchronization actions on locks. The "*Lock Acquire*" request changes the lock's status from "*unlocked*" to "*locked*" and the "*Lock Release*" request changes the status from "*locked*" to "*unlocked*".

We consider the dynamic buffer allocation under two situations by the dependency of the two simultaneously coming requests. Here, "dependent" means the two simultaneous requests access the same lock. In contrast, "independent" means the two requests access different locks. The following paragraphs and Fig. 4 illustrate synchronization actions under different cases.

Situation I: only one request or two independent requests come.

A: When a "*Lock Acquire*" request comes,

1) If the requested lock's status is "*unlocked*", the request goes along the bypass path to access the Synchronization Variable Pool directly. The "*successful*" acknowledgement is sent back to the source node. In the meantime, the lock's status is changed into "*locked*". [see (3) and (5) in Fig. 4].
2) If the requested lock is on its "*locked*" status, (i) if the buffer queues in the physical buffers are full, the "*failed*" acknowledgement is sent back to the source node, and (ii) if the buffer queues are not full, the request is buffered in the related virtual buffer until the lock is released. The lock's status remains. [see (3) in Fig. 4]

B: When a "*Lock Release*" request comes,

1) If there are one or more "*Lock Acquire*" requests buffered in the virtual buffer which is related to the to-be-released lock, the "*Lock Release*" request doesn't need to access the Synchronization Variable Pool but the first "*Lock Acquire*" request in the virtual buffer is forwarded into the Synchronization Variable Pool. The related "*successful*" acknowledgement is sent back. The lock's status remains. [see (5), (6) and (7) in Fig. 4].
2) If the related virtual buffer is empty, the "*Lock Release*" request goes along the bypass path to access the Synchronization Variable Pool and changes the lock's status to be "*unlocked*". [see (6) in Fig. 4]

Situation II: Two dependent requests come simultaneously.

A: When two dependent "*Lock Acquire*" requests come,

1) If the requested lock's status is "*unlocked*", one of the two requests is selected to access the Synchronization Variable Pool via the bypass path directly and the related "*successful*" acknowledgement is sent back. If the buffer queues are full, the "*failed*" acknowledgement is sent back to the node sending the other request. If not, the other request is buffered in the related virtual buffer. Meanwhile, the lock's status is changed into "*locked*". The selection conforms to the round-robin policy. [see (2) in Fig. 4]
2) If the requested lock is on its "*locked*" status, (i) if the buffer queues are full, two "*failed*" acknowledgement are sent back to where the two requests are from, respectively, (ii) if there is only one empty item in the buffer queues, one request is buffered and a "*failed*" acknowledgement is sent back to the node sending the other request, and (iii) if there are at least two empty items in the buffer queues, the two requests are buffered in their related virtual buffers. The lock's status remains. [see (4) in Fig. 4]

B: When the dependent "*Lock Acquire*" request and "*Lock Release*" request come,

1) If there are one or more "*Lock Acquire*" requests buffered in

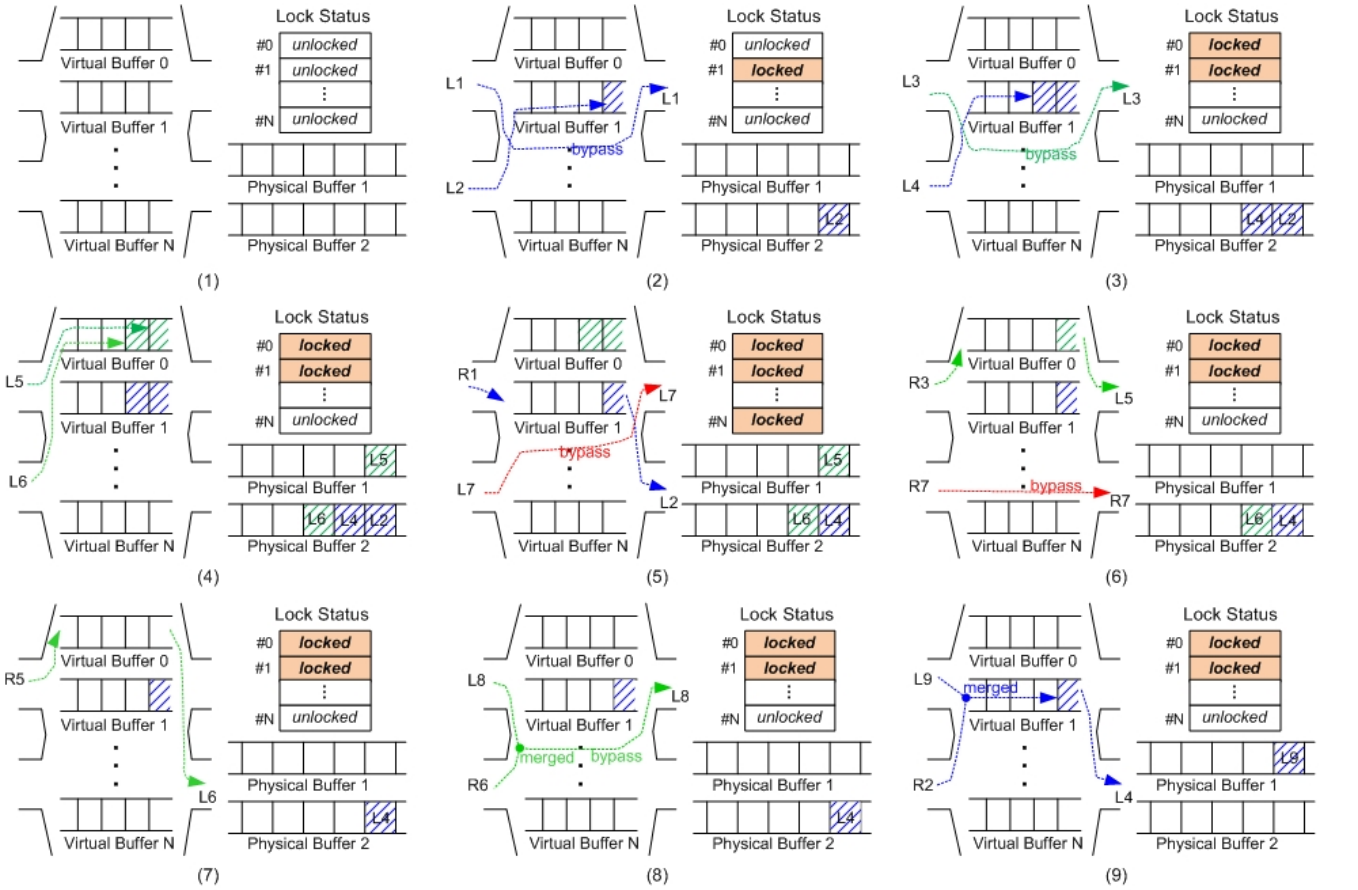

Fig. 4. Different cases of performing synchronization actions

the related virtual buffer, the "*Lock Release*" request doesn't need to access the Synchronization Variable Pool, the first "*Lock Acquire*" request in the virtual buffer is forwarded into the Synchronization Variable Pool, the related "*successful*" acknowledgement is back, and the coming "*Lock Acquire*" request goes into the virtual buffer. The lock's status remains. [see (9) in Fig. 4]

2) If the related virtual buffer is empty, the "*Lock Release*" request doesn't need to access the Synchronization Variable Pool, while the "*Lock Acquire*" request goes along the bypass path to access the Synchronization Variable Pool directly and the related "*successful*" acknowledgement is sent back. The lock's status remains. [see (8) in Fig. 4]

To help understand Fig. 4, we take (2), (4) and (9) in Fig. 4 as examples to explain the dynamic buffer allocation in detail. In (2), when L1 and L2 comes concurrently, Lock #1 is "*unlocked*". L1 acquires Lock #1 successfully and hence the status of Lock #1 changes to be "*locked*", but L2 is stored in Physical Buffer 2 (in Virtual Buffer 1 logically). In (4), when L5 and L6 comes concurrently, Lock #0 is "*locked*". L5 is stored in Physical Buffer 1 because it's from the processor and L6 is stored in Physical Buffer 2 because it's from Physical Buffer 2. However, they are both in Virtual Buffer 0 logically. In (9), when L9 and R2 comes concurrently, Lock #1 is "*locked*" and L4 is in Physical Buffer 2 (in Virtual Buffer 1 logically) to wait for acquiring Lock #1. L4 acquires Lock #1 successfully as soon as R2 releases Lock #1. L9 goes into Physical Buffer 1 (in Virtual Buffer 1 logically).

TABLE I
SYNTHESIS RESULTS

	N=2		N=8		N=32		N=128	
	A	**F**	**A**	**F**	**A**	**F**	**A**	**F**
Q= 1	0.89	1.79	1.18	1.60	1.93	1.47	4.44	1.28
Q= 2	1.78	1.56	2.26	1.42	3.32	1.32	5.11	1.20
Q= 4	3.69	1.43	4.04	1.30	5.40	1.21	8.32	1.16
Q= 8	7.60	1.32	8.39	1.22	9.31	1.16	13.41	1.12
Q=16	14.63	1.16	18.41	1.11	20.76	1.06	23.77	1.02
Q=32	29.02	1.04	38.64	0.99	42.12	0.95	50.46	0.92

A: Area (Kilo gate); **F:** Frequency (GHz)

C. Hardware Implementation

The SH design is synthesized using Synopsys® Design Compiler in Chartered® 0.13 μm technology. The synthesis results are listed in TABLE I. For all, the clock frequency can reach more than 900 MHz. As **Q** and **N** increases, the area cost increases and the clock frequency decreases, because larger **Q** and **N** result in more storage and logic area and less clock frequency. For instance, when **Q**=1and **N**=2, the area is 0.89K and it can reach 1.79 GHz. When **Q**=32 and **N**=128, the area rises up to 50.46K and the frequency reduces to 0.92 GHz. The values of **Q** and **N** affect not only the hardware cost but also the performance. Although the increase of **Q** and **N** leads to larger hardware cost, the designs gain better performance (see in Section IV).

IV. Experiments and Results

We perform experiments to evaluate the SH in a multi-core NoC platform, applying both synthetic and application workloads.

TABLE II
DEFINITIONS AND NOTATIONS

Q:	The depth of buffer queue in each physical buffer
N:	The number of locks
C:	Average cycles of acquiring a lock successfully in a simulation. **C** = (the time when the first acquire is sent in a simulation - the time when the last "*successful*" acknowledgment is received in a simulation) / clock period
L:	Lock life time: when a lock is acquired by a PM node successfully and will be released after **L** cycles.
$\gamma(t)$**:**	Buffer utilization at cycle #t. $\gamma(t)$ = the number of used items in the two physical buffers at cycle #t / the total item numbers in the two physical buffers (2***Q**).
Γ**:**	Average buffer utilization during a simulation. Γ = the sum of $\gamma(t)$ / the total cycles in a simulation.

A. Experimental Platform

We constructed a multi-core NoC experimental platform as shown in Fig. 1. The multi-core NoC uses the LEON3 [10] as the processor in each PM node and uses the Nostrum NoC [11] as the on-chip network. The LEON3 processor core is a synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture. The Nostrum NoC is a 2D mesh packet-switched network with configurable size. It serves as a customizable platform.

B. Definitions and Notations

To facilitate the analysis and discussion in the following subsections, we first define a set of symbols in TABLE II.

C. Simulation Results with Synthetic Workloads

First, we use synthetic workloads to evaluate the SH's performance. In the experiment, all nodes acquire the same lock located in the central node. If a node receives a "*failed*" acknowledgment from the central node, it re-acquires the lock again until it receives a "*successful*" acknowledgment. Once a node acquires the lock successfully, it holds the lock for **L** cycles, and then releases it. The simulation ends after all nodes acquire the lock successfully and release it.

Cost of Successful Synchronization

In this subsection, the cost of successful synchronization is discussed with respect to three factors: (i) network size, (ii) depth of buffer queue in the two physical buffers, and (iii) lock life time. The cost of successful synchronization is reflected by the average cycles of acquiring a lock successfully (**C**). It is demonstrated that our SH design achieves higher performance as either the network size or the depth of the buffer queue in the two physical buffers is becoming larger.

Fig. 5(a) plots the average cycles of acquiring a lock successfully (**C**) versus the network size and the queue depth. the lock life time (**L**) is fixed to be 10 cycles. We can see that (i) as the network size increases, the **C** increases, because network latency becomes larger, and (ii) for the same network size, the larger the queue depth is, the less the **C** is.

Fig. 5(b) plots the average cycles of acquiring a lock successfully (**C**) versus the network size and the lock life time (**L**). The queue depth is fixed to be 8. We can observe that (i) with the increase of the network size, the **C** is increasing, and (ii) for the same network size, the **C** is positively proportional to the **L**. Larger **L** leads to longer waiting time for lock release, so more packets transmit in the network and hence network latency becomes larger, under the same queue depth condition.

Analysis of Buffer Utilization

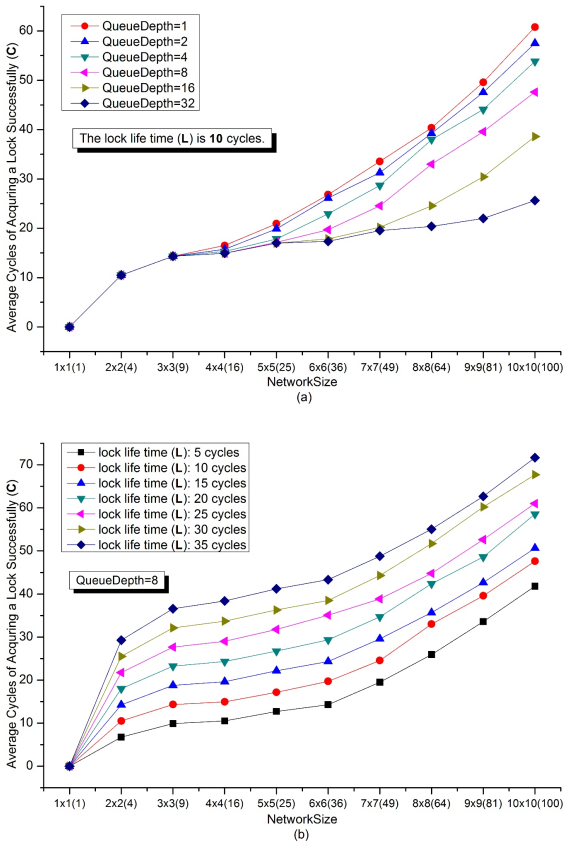

Fig. 5. Cost of Successful Synchronization

Since the SH's main feature is the two physical buffers and their "virtual buffer" organization, the buffer utilization is analyzed. In the experiment, The buffer utilization ($\gamma(t)$) and the average buffer utilization (Γ) of the SH in the central node are obtained.

Fig. 6 shows the average buffer utilization (Γ) of the SH in the central node versus the lock life time and the queue depth. The network size is fixed to be 8x8(64). As we can see, (i) the Γ for larger queue depth is lower than that for smaller queue depth. For instance, the Γ for QueueDepth=1, 2, 4, 8, 16 and 32 are more or less 97%, 96%, 94%, 90%, 76% and 47%, and (ii) the Γ is almost a constant for each value of queue depth, no matter what the lock life time is. This is because larger lock life time leads to not only longer simulation time but also the larger sum of $\gamma(t)$.

Fig. 7 shows the buffer utilization ($\gamma(t)$) of the SH in the central node for different queue depth, versus the clock cycle. For larger queue depth, the width of the bottom of the curve is smaller and the top of the curve is narrower. The width of the bottom reflects how long the simulation runs, so the smaller the width of the bottom is, the better the performance is. The narrower top of the curve leads to the lower Γ. For instance, for QueueDepth **Q**=1, the width of the top is almost equal to the width of the bottom. It means that the two physical buffers are always close to be full. For QueueDepth=32, the top is very sharp, meaning that the two physical buffers are only close to be full during a very small part of the entire simulation. This is why the (Γ) for larger queue depth in Fig. 6 is lower. We can obtain that higher queue depth can cause better performance, but brings more area cost and lower (Γ).

D. Simulation Results with Application Workloads

This subsection evaluates the SH's performance in terms of speedup by employing two synchronization-intensive applications: Livermore Loop 6 and Wavefront Computation. In Livermore

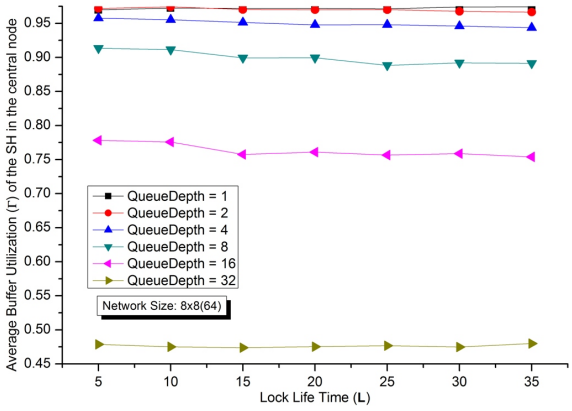

Fig. 6. average buffer utilization (Γ) of the SH in the central node

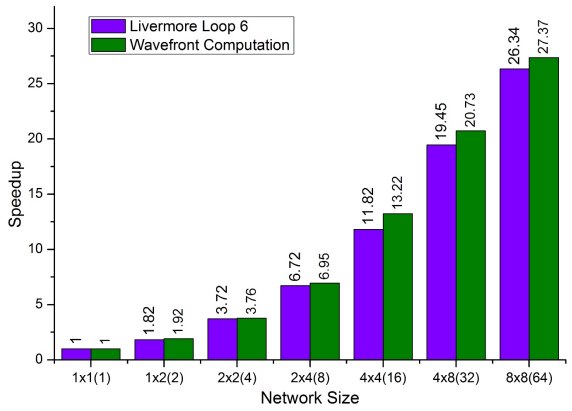

Fig. 8. Speedup of Livermore Loop 6 and Wavefront Computation

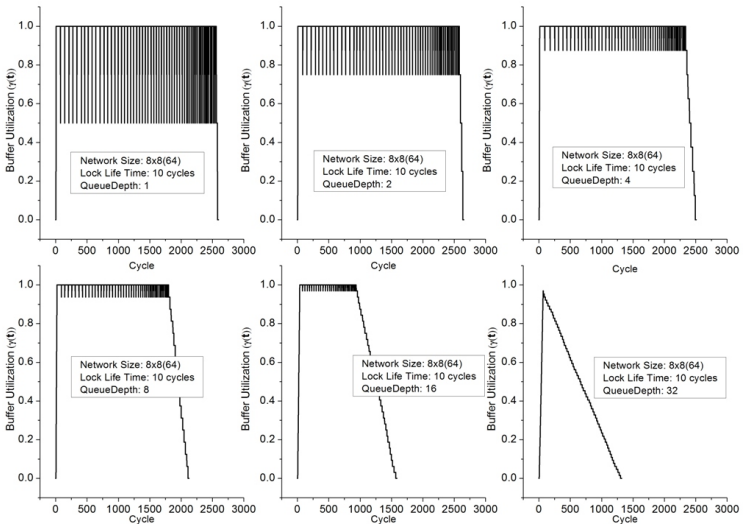

Fig. 7. buffer utilization ($\gamma(t)$) in the central node for different queue depth

Loop 6, the computation in each iteration depends on the values calculated in all previous iterations. In Wavefront Computation, the computation of each matrix element depends on its neighbors to the left, above, and above-left.

Fig. 8 shows the application speedup results versus the network size. We can see that our SH design achieves fairly good speedup. When the network size increases, the speedup ($\Omega_m = T_{1node}/T_{mnode}$, where T_{1node} is the single PM node execution time as the baseline, T_{mnode} the execution time of m PM node(s).) goes up from 1, 1.82, 3.72, 6.72, 11.82, 19.45 to 26.34 for the Livermore Loop 6 and from 1, 1.92, 3.76, 6.95, 13.22, 20.73 to 27.37 for the Wavefront Computation. To make the comparison fair, we calculate the per-node speedup by Ω_m/m. As the network size increases, the per-node speedup decreases from 1, 0.91, 0.93, 0.84, 0.74, 0.61 to 0.41 for the Livermore Loop 6 and from 1, 0.96, 0.94, 0.87, 0.83, 0.65 to 0.43 for the Wavefront Computation. This means that, as the network size increases, the speedup acceleration is slowing down. This is due to that the communication latency goes up nonlinearly and the synchronization overhead increases with the network size, limiting the performance.

V. Conclusion

In this paper, we have proposed a hardware Synchronization Handler (SH) on multi-core NoCs with distributed memories to enable efficient shared variable synchronization. The SH features two physical queues to buffer and handle synchronization requests issued by the local and remote processors via the on-chip network. The two physical buffers are dynamically allocated and logically organized as multiple virtual buffers. This dynamic buffer allocation and virtual organization results in efficient buffer utilization and gains in performance. The SH design has been implemented and integrated in our cycle-accurate multi-core NoC platform. To quantify its speed and area, the SH design was synthesized in 130nm technology under various queue dimensions, showing that it can run more than 900 MHz with small area overhead. Finally, both synthetic and synchronization-intensive application workloads have been applied to evaluate the cost of successful synchronization, the utilization of dynamically allocated buffers and application speedup.

Acknowledgment

The research is partially supported by the FP7 EU project MOSART (No. IST-215244), the National 863 Program of China (No. 2009AA011704), the Innovative Team of High-performance Microprocessor Technology (No. IRT 0614), and the National Natural Science Foundation of China (No. 60676010).

References

[1] A. Jantsch and H. Tenhunen, *Networks on chip*. Kluwer Academic Publishers, 2003.

[2] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," *ACM Comp. Surveys*, vol. 38, no. 1, pp. 1–51, Mar. 2006.

[3] P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh, "Performance evaluation and design trade-offs for network-on-chip interconnect architectures," *IEEE Trans. on Computers*, vol. 54, no. 8, pp. 1025–1040, Aug. 2005.

[4] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, "Power/performance hardware optimization for synchronization intensive applications in mpsocs," in *Proc. of the Conf. on Design, automation and test in Europe (DATE'06)*, 2006, pp. 606–611.

[5] J. Sampson, R. Gonzalez, J.-F. Collard, N. Jouppi, and M. Schlansker, "Fast synchronization for chip multiprocessors," *ACM Computer Architecture News*, vol. 33, no. 4, pp. 64–69, Apr. 2005.

[6] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, "Efficient synchronization for embedded on-chip multiprocessors," *IEEE Trans. on VLSI*, vol. 14, no. 10, pp. 1049–1062, Oct. 2005.

[7] O. Villa, G. Palermo, and C. Silvano, "Efficiency and scalability of barrier synchronization on noc based many-core architectures," in *Proc. of the 2008 Int'l Conf. on Compilers, architectures and synthesis for embedded systems (CASES'08)*, 2008, pp. 81–89.

[8] W. Zhu, V. Sreedhar, Z. Hu, and G. Gao, "Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures," in *Proc. of the 34th annual Int'l Symp. on Computer Architecture (ISCA'07)*, 2007, pp. 35–45.

[9] C. Yu and P. Petrov, "Distributed and low-power synchronization architecture for embedded multiprocessors," in *Proc. of the 6th IEEE/ACM/IFIP Int'l Conf. on Hardware/Software codesign and system synthesis (CODES+ISSS'08)*, 2008, pp. 73–78.

[10] "Leon3 processor," in *http://www.gaisler.com*.

[11] A. Jantsch *et al.*, "The nostrum network-on-chip," in *http://www.ict.kth.se/nostrum*.