IEICE TRANS. INF. & SYST., VOL.E98–D, NO.2 FEBRUARY 2015

LETTER *Special Section on Reconfigurable Systems*

Distributed Synchronization for Message-Passing Based Embedded Multiprocessors

Hao XIAO[†a)], ***Member***, **Ning WU**[†], **Fen GE**[†], **Guanyu ZHU**[††], ***Nonmembers***, ***and*** **Lei ZHOU**[†], ***Student Member***

SUMMARY This paper presents a synchronization mechanism to effectively implement the *lock* and *barrier* protocols in a decentralized manner through explicit message passing. In the proposed solution, a simple and efficient synchronization control mechanism is proposed to support queued synchronization without contention. By using state-of-the-art Application-Specific Instruction-set Processor (ASIP) technology, we embed the synchronization functionality into a baseline processor, making the proposed mechanism feature ultra-low overhead. Experimental results show the proposed synchronization achieves ultra-low latency and almost ideal scalability when the number of processors increases.

key words: *embedded multiprocessors, synchronization, message-passing, application-specific instruction-set processor*

1. Introduction

In multiprocessors, the synchronization overhead, which includes the execution time, communication latency and traffic contention, is known to have a significant impact on the system's performance and scalability. Conventional synchronization implementations that utilize the atomic *read-modify-write* instructions are known to suffer from serious traffic contention caused by remote polling [1]. Therefore, most recent multiprocessors, like Tilera [2], use cache-coherent approach on top of shared memory, which allows polling to be performed at local cache. However, such a solution does not resolve the power problem caused by polling. Moreover, significant bus contention also occurs when a processor releases the synchronization variable, which leads to invalidations and subsequent misses in all remote caches [3]. In order to address these problems, a hardware-supported solution is proposed in [4], where a centralized hardware engine monitors all synchronization requests globally and serves them in a first-in-first-out (FIFO) order. In our previous work [5], [6], two centralized hardware engines are proposed to support the synchronization for shared-memory and message-passing MPSoC architectures, respectively. Our recent work [7] further optimizes the centralized controller in [6] by enabling it to support fully queued *lock* synchronization. However, these centralized solutions fundamentally introduce a bottleneck when the system scales to more cores. Therefore, the primary objective of this work is to exploit *distributed* models performing synchronization in a *decentralized* manner, rather than using any centralized engine. Moreover, experimental comparisons to centralized hardware approaches are analyzed in aspects of performance, scalability and area efficiency. In [3], a distributed synchronization model is proposed, where each processor locally maintains a state of all synchronization variables and participates in decentralized protocol. However, this method is only applicable to shared-bus architecture, which fundamentally limits its scalability and constrains its feasibility in multiprocessor architectures.

In this paper, we present a distributed synchronization model for message-passing based multiprocessors. Unlike centralized methods that employ dedicated controllers to manage all synchronization variables globally, the proposed architecture distributes each variable to each single unique processor. Local to each processor, a synchronization controller maintains a precise state of the synchronization variable assigned to it. To acquire and release a certain synchronization variable, one processor explicitly sends a request to the base processor, which either grants the access or rejects it, and then queues the requester. We implement the proposed synchronization mechanism using the ASIP methodology. Experimental results show the proposed synchronization achieves ultra-low latency and almost ideal scalability when the number of processors increases.

This paper is organized as follows. Section 2 illustrates the proposed synchronization protocol, which is followed by the hardware implementation and evaluation results in Sect. 3. Finally, conclusions are drawn in Sect. 4.

2. Distributed Synchronization Protocol

This paper focuses on two most common synchronization primitives: *lock* and *barrier*, which can be used as basic blocks of most software synchronization routines. The *lock* provides a *mutual exclusion* which allows only a single processor to hold a *lock* at any one time, and the *barrier* is to force a rendezvous of all processors.

2.1 *Lock* Synchronization Protocol

In the proposed synchronization model, each *lock* variable is assigned a single unique processor. Local to each processor, two registers, QUEUE and AHEAD, are used to queue the *lock* requests, whose management algorithms are illustrated in Figs. 1 and 2 respectively. The QUEUE is an *N* bit

Manuscript received April 30, 2014.
Manuscript revised July 29, 2014.

[†]The authors are with the College of Electronic and Information Engineering, Nanjing University of Aeronautics and Astronautics, Nanjing, 210016 China.
[††]The author is with the Shannon Lab, Huawei, China.
a) E-mail: xiaohao@nuaa.edu.cn

DOI: 10.1587/transinf.2014RCL0001

```
1) if (acquire-request-received)
2)       QUEUE = QUEUE | PE_ID;
3) if (release-request-received)
4)       QUEUE = QUEUE & ~PE_ID;
```

Note:
** QUEUE is an N bit vector, where N is the num. of total processors.*
*** PE_ID is the vector ID of the processor node that is sending the request, e.g. node 3's vector ID is 0b100.*

Fig. 1 *Lock* QUEUE management in base processor.

```
lock_acquire(){
1) send acquire request;
2) if (grant) {hold lock and continue;}
3) if (reject) {
4)       receive AHEAD from base;
5)       suspend;}}

if(release_notification_received) {
1) AHEAD = AHEAD − 1;
2) if (AHEAD == 0) {
3)       send lock_retry request;
4)       receive grant then continue;}
5) else {keep suspending;}}
```

Fig. 2 AHEAD management in local processor.

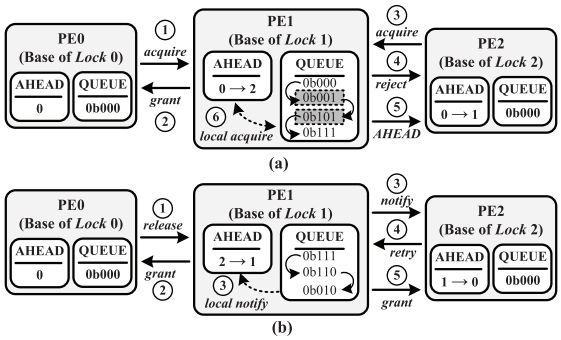

Fig. 3 *Lock* synchronization protocol: (a) acquire *lock*, (b) release *lock*.

register, where N is the number of processor nodes. It represents the existing of processors currently holding and pending on a specific *lock*. Thus, an acquisition request sets the requester's vector bit to 1, and a release request, by contrast, clears the *lock* holder's vector bit. The AHEAD is a counter, whose maximum value is N. It is local to each processor for indicating the number of processors have requested the same *lock* before it. Its initial value is zero, which allows its local processor to attempt the acquisition. A positive response enables the requester hold the *lock* with its AHEAD unchanged. Whereas, a negative reply forces the requester to suspend and update its AHEAD according to the following value from the base processor. When a holder releases its *lock*, the base informs all the processors pending on this *lock*, making their local AHEAD decrease by one. Then, if someone's AHEAD becomes zero, it resumes immediately to retry the *lock*. Otherwise, it keeps on suspending.

Figure 3 (a) illustrates a scenario whereby three processors use the proposed model to acquire *lock* 1 sequentially. *Transition* 1 represents PE0 acquires first by sending a request to *lock* 1's base processor PE1. Since the QUEUE is initially empty, PE1 grants the acquisition (*Transition* 2),

```
1) if (QUEUE==constant vector) { /* reach barrier */
2)       QUEUE = 0;
3)       inform remote PEs;}
4) else {
5)       QUEUE = QUEUE | PE_ID;
6)       response reject;}
```

Note:
** QUEUE is an N bit vector, where N is the num. of total processors.*
*** PE_ID is the vector ID of sending processor node, e.g. node 3's vector ID is 0b100.*

Fig. 4 *Barrier* QUEUE management in base processor.

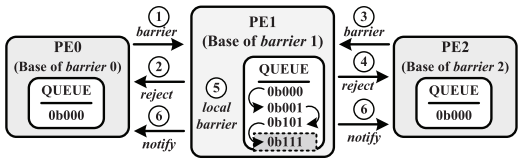

Fig. 5 *Barrier* synchronization protocol.

and meanwhile sets PE0's vector bit, 0b001, in QUEUE. After PE0, PE2 acquires *lock* 1 sequentially (*Transition* 3). Instead of grant, a reject is replied, because there has been queued request already. In this case, the base processor further sends a new AHEAD to PE2 (*Transition 5*), whose value equals the sum of all bits in current QUEUE, and also sets PE2's vector bit in QUEUE. After being rejected, PE2 suspends and updates its local AHEAD according to the received value. At last, the base processor, PE1, acquires *lock* 1 and deals with its local request internally (*Transition 6*). Since *lock* 1 is still unavailable at this time, PE1 sets itself in the QUEUE, updates its local AHEAD and then suspends.

Figure 3 (b) illustrates the scenario whereby PE0 hands over *lock* 1. In *Transition* 1/2, PE0 sends a release request to PE1, which grants immediately enabling PE0 to proceed. Meanwhile, PE1 removes PE0 from the QUEUE and informs this release to all pending PEs (*Transition* 3). It is noteworthy that PE1 signals itself internally rather than sending messages. The release notifications make PE1 and PE2, both of which are pending on *lock* 1, decrease their local AHEAD by 1. Then, PE 2's AHEAD becomes 0, and thus retries *lock* 1 immediately (*Transition* 4/5). By contrast, PE1 keeps on suspending

2.2 *Barrier* Synchronization Protocol

In the proposed synchronization model, *barrier* is also well-supported in a way similar to the *lock*. Figure 4 illustrates the functionality required to maintain the *barrier* QUEUE and Fig. 5 shows a *barrier* synchronization scenario using the proposed model. In Fig. 5, *Transition* 1 to 4 represent PE0 and PE2 reach the *barrier* successively and send their *barrier* requests to the base processor respectively. Since the last PE has not arrived yet, the base processor rejects the requests, and sets the vector bits of arrived PEs in the QUEUE. When PE1 finally arrives (*Transition* 5), all required PEs have reached the *barrier*, which is indicated by the condition that current QUEUE equals a decided constant vector. Then, the QUEUE is cleared to zero, and all the

pending processors, whose vector IDs are previously saved in QUEUE, are signaled to resume (*Transition* 6). It is noteworthy that, unlike the *lock* protocol, *barrier* protocol is not sensitive to the arriving order, and thus the AHEAD register is unused.

3. Implementation and Evaluation

3.1 Implementation

We implement the proposed synchronization model in our hybrid shared-memory/message-passing MPSoC architecture [6]. As shown in Fig. 6, this architecture consists of two communication domains, shared-memory and message-passing. The former connects all the processor nodes via a conventional bus enabling them to have a shared address space. The latter offers the MPSoC another on-chip network to support fast inter-processor communication through point-to-point message passing. Our proposed synchronization model is built on top of the message-passing domain, where all synchronization requests and responses are sent through explicit messages. The synchronization protocol proposed in Sect. 2 is implemented in each processor by using the state-of-the-art ASIP methodology, LISA [8].

The proposed synchronization model is programmed according to our ASIP programming model [6]. A custom instruction, *msg_send*, is used to implement the synchronization request, which issues a single-word (32 bit) message to a specified base processor. The synchronization message consists of three portions, 16-bit *lock*/*barrier* ID, 8-bit requester processor ID and 8-bit synchronization control token, *e.g. lock* acquisition or *lock* release.

3.2 Experimental Result

In this section, we present our experiments to evaluate the performance of the proposed *lock* and *barrier* mechanisms. For comparison, we also implement three other synchronization models, one is the conventional *polling* model and the other two are *centralized contention-free* models for *shared-memory* (hereinafter, "*SM*") [5] and *message-passing* (hereinafter, "*MP*") [6] architectures, respectively.

First, to evaluate the performance of the *lock*, we use two micro-benchmarks introduced in [9]. In one benchmark, one processor acquires and holds the *lock* for a long time while all the other processors request the *lock* and find it busy. Then the holder finally releases the *lock* and the benchmark begins, continuing until all processors are able to acquire and release the *lock* a single time. This benchmark measures the latency of the *lock* by letting each processor to merely acquire the *lock* and release it immediately. The other is a less contrived high-contention micro-benchmark, which accesses the critical section a total of 3,200 times; these accesses are distributed evenly among the processors. Once in the critical section, a processor waits 800 cycles before releasing the *lock* (this stall simulates access to, and computation of, protected data). After release, the releasing processor waits for a random time selected from a uniform distribution. The mean of the distribution is five times the critical section delay (4,000 cycles).

Figure 7 (a) shows the result of the first benchmark with the increasing core counts of multiprocessors. In this figure, a flat horizontal curve corresponds to ideal scalability — a *lock* with performance independent of core count. We can see that the proposed one is superior to all others in every aspect: the scalability is the best and the absolute latency is the lowest. In more detail, the three contention-free *lock* models, the proposed one and the two centralized ones, achieve almost ideal scalability. Whereas, the *polling lock* scales poorly, which is due to the traffic contention caused by the remote polling. Considering the latency, the *SM-lock* is the worst, because it has to use interrupt mechanism for handing over the *lock* token. The *polling lock* is originally quite fast. However, due to the poor scalability, its latency grows rapidly as the core count increases. Both the proposed *lock* and the *MP-lock* use ASIP method on message-passing architecture. Thanks to the decentralized mechanism, the proposed one exhibits lower latency than the centralized counterpart. Figure 7 (b) shows the execution time of the second benchmark, where the *polling lock* scales poorly, caus-

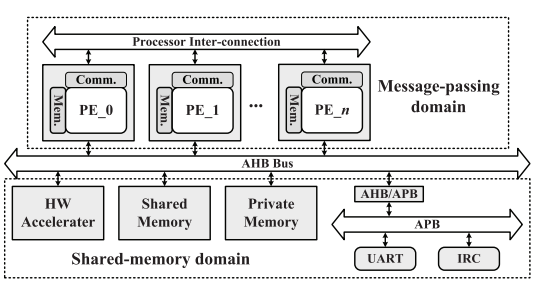

Fig. 6 Hybrid shared-memory/message-passing MPSoC architecture.

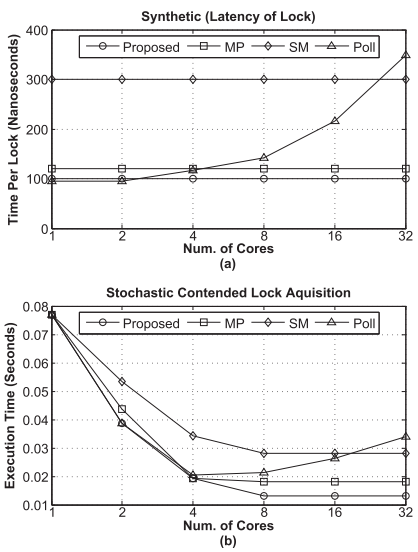

Fig. 7 Performance of *lock* benchmarks: (a) benchmark of *lock* latency, (b) benchmark of high-contention *lock* scenario.

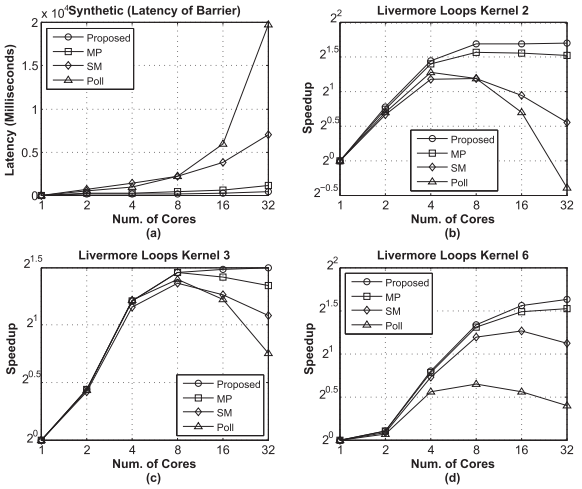

Fig. 8 Performance of *barrier* benchmarks: (a) Synthetic, (b) Livermore loops Kernel 2, (c) Livermore loops Kernel 3, (d) Livermore loops Kernel 6.

Table 1 Area comparison of multiple synchronization models.

Architecture	Proposed		Centralized	
	Baseline	ASIP	SM [6]	MP [7]
Area (μm^2)	103637	106080	59514	58142
	2443			

ing its throughput to degrade rapidly. While using the three contention-free *locks*, the execution time becomes constant after the core count exceeds eight, which corresponds to desirable *lock* throughput independent of core count. Among these three models, the proposed *lock* further shows the best performance, which is due to its lowest latency.

Figure 8 shows the performance of the four *barrier* models with four benchmarks: a synthetic benchmark and three kernels from Livermore loops (Kernel 2, 3, 6). For more details of these benchmarks, please refer to our previous work [6]. As shown in Fig. 8 (a), the latency of the *polling barrier* and the *SM-barrier* grows rapidly as the core count increases. While the latency of the two ASIP-based *barriers* vary only a little, which exhibits much better scalability. Regarding the kernels shown in Fig. 8 (b)–(d), using the proposed *barriers* lead to a constantly increased speedup as the size of multiprocessor scales. Whereas, the kernels using the *polling barrier* and *SM-barrier* only speed up at the initial phase, and then start to degrade from a certain point. This is because the performance degradation caused by contention or interrupt overhead finally overwhelms the speedup gained from parallel processing. Considering the two ASIP-based *barriers*, the proposed one is better, which is due to the lower latency and better scalability caused by the proposed decentralized mechanism.

Finally, we evaluate the area efficiency of the proposed synchronization. Table 1 lists the area of the proposed ASIP and its baseline processor, as well as the other two centralized controllers [5], [6], under TSMC 90nm technology at 200MHz. As shown, the proposed synchronization mechanism costs only 2443 μm^2 additional area per node, which is much smaller than the other two centralized counterparts. This area saving is due to that the proposed mechanism simplifies the synchronization control logic and avoids the using of FIFO memory.

4. Conclusion

This paper presents an efficient synchronization mechanism for message-passing based multiprocessors. By implementing the synchronization protocol in a completely distributed manner, the proposed solution improves the synchronization performance and scalability significantly. Furthermore, the logic control of this solution is straightforward, which eases the hardware implementation significantly. Finally, our experiments confirm the effectiveness of the proposed approach, which achieves ultra-low latency and almost ideal scalability when the number of processors increases.

Acknowledgments

The authors would like to thank the partial support from Natural Science Foundation of Jiangsu Province BK20140834, Huawei Innovation Research Program and National Natural Science Foundation of China 61376025 61106018.

References

[1] D.E. Culler, J.P. Singh, and A. Gupta, Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufmann, 1998.
[2] Tilera Corporation, Tile Processor Architecture Overview for the TILEPro Series, 2011.
[3] C. Yu and P. Petrov, "Low-cost and energy-efficient distributed synchronization for embedded multiprocessors," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.18, no.8, pp.1257–1261, Aug. 2010.
[4] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, "Efficient synchronization for embedded on-chip multiprocessors," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.14, no.10, pp.1049–1062, Oct. 2006.
[5] H. Xiao, T. Isshiki, A.U. Khan, D. Li, H. Kunieda, Y. Nakase, and S. Kimura, "A low-cost and energy-efficient multiprocessor system-on-chip for UWB MAC layer," IEICE Trans. Inf. & Syst., vol.E95-D, no.8, pp.2027–2038, Aug. 2012.
[6] H. Xiao, T. Isshiki, D. Li, H. Kunieda, Y. Nakase, and S. Kimura, "Optimized communication and synchronization for embedded multiprocessors using ASIP methodology," IPSJ Trans. System LSI Design Methodology, vol.5, pp.118–132, Aug. 2012.
[7] H. Xiao, T. Isshiki, D. Li, and H. Kunieda, "Efficient synchronization for message-passing based embedded multiprocessors," 2014 Int. Conf. Information and Communication Technology for Embedded Systems (IC-ICTES), Jan. 2014.
[8] LISA 2.0, Synopsys Inc., http://www.synopsys.com/
[9] F. Heinlein, "Optimized multiprocessor communication and synchronization using a programmable protocol engine," Technical Report, CSL-TR-98-759, Stanford University, March 1998.