Journal of Parallel and Distributed Computing
Manuscript Draft

Manuscript Number: JPDC-11-296

Title: Efficient Synchronization with the Rendezvous Mechanism for Multi-core System.

Article Type: Regular Paper

Keywords: Multiprocessor synchronization; Multiprocessor system-on-chip; synchronization; network-on-chip.

Abstract: The traditional synchronization, polling and barriers, incur heavy network utilization and must tolerate the worst-case thread due to the inflexible mechanism. As the number of processors increases, these approaches lack scalability and lead to bottleneck performance. This paper proposes a rendezvous mechanism that only has constant network transactions, and provides the flexible synchronization by defining the dependent relationships based on applications. The proposed management develops two main operations for flexible synchronization. 1.This management permits that cores, which have completed threads and do not have the relationships of data dependence, proceed without tolerating the worst-case thread. 2. This management provides the forward execution when a thread must wait due to data dependence. The simulation results demonstrate that the rendezvous mechanism can develop efficient communication for synchronization, and significantly decreases network utilization to improve the scalability and performance of multi-core systems.

Highlights

>This paper proposed the flexible synchronization based on the thread-level dependent for multi-core systems. >1. This management permits processing elements to proceed without tolerating the worst-case thread. >2. This management provides the forward execution if an element must wait due to dependence.

Efficient Synchronization with the Rendezvous Mechanism for Multi-core System

Jih-ching Chiuand Kai-ming Yang[1]
[1] Electrical Engineering Department, National Sun Yat-Sen University, Taiwan
d953010024@gmail.com Tel: 886-7-5252000 ext. 4183

***Abstract*—The traditional synchronization, polling and barriers, incur heavy network utilization and must tolerate the worst-case thread due to the inflexible mechanism. As the number of processors increases, these approaches lack scalability and lead to bottleneck performance. This paper proposes a rendezvous mechanism that only has constant network transactions, and provides the flexible synchronization by defining the dependent relationships based on applications. The proposed management develops two main operations for flexible synchronization. 1.This management permits that cores, which have completed threads and do not have the relationships of data dependence, proceed without tolerating the worst-case thread. 2. This management provides the forward execution when a thread must wait due to data dependence. The simulation results demonstrate that the rendezvous mechanism can develop efficient communication for synchronization, and significantly decreases network utilization to improve the scalability and performance of multi-core systems.**

***Index Terms*—Multiprocessor synchronization; Multiprocessor system-on-chip; synchronization; network-on-chip.**

I. Introduction

Multi-processor systems integrate simpler cores into a single System-on-Chip (SoC), replacing the single complex core in embedded systems to achieve high performance. Therefore, efficient inter-processor communication and synchronization become significant influences on throughput. The popular synchronization, barriers and spinlocks, is always used to protect a critical section based on a shared variable. However, the larger-scale multiprocessors could incur high-contention situations before resulting in throughput bottleneck because the spinlocks, implemented by polling, effectuate much utilization of interconnection, and the barriers, using an inflexible coherence mechanism, have a potentially greater latency.

To protect the critical section, traditional approaches utilized polling to perform *spinlocks*, such as *test-&-set*, *test-test-&-set*, *exponential backoff*, *ticket*, and *queue* [18] [19] [22]. *Test-&-set* is the simplest procedure for synchronization because it executes a consecutive read-modify-write atomic operation to implement a mutual exclusion lock. *Test-test-&-set* lock is an alternative to the *test-&-set* lock. When *test-&-set* is in a busy lock, this operation does not spin the shared variable to reduce network contention. The *ticket* lock method implements the test-&-increment instruction to count the number of requests for acquiring the lock and the number of releases from the lock. In this manner, these operations still incur a great amount of remote access when parallel threads are waiting for a long time period. Considering scalability, the remote access also causes severe network contention for synchronization. Reducing polling frequency is the most direct approach to decrease the opportunity of conflict. *Test-test-&-set* locks reduce the polling frequency by the different time slots based on *exponential backoff*. In contrast, due to the unpredictability of duration of executing threads, this mechanism may not immediately obtain a freed critical section. The *queuing* lock method, combined with the compare-&-swap operation, follows the First-In-First-Out (FIFO) order to hold the waiting threads. The approach is one of the more efficient operations, and is also used in other designs for synchronization [5][16][20][22][24]. This proposed mechanism has an excellent performance achievement and network contention. However, in [7], they analyzed the performance of communication patterns based on the dependence of threads. We found that the dependence among parallel threads is also another significant factor affecting the throughput in a multi-core system. The conventional FIFO order would not develop a flexible mechanism for these communication patterns. Instead, when the workload of parallel programs cannot be segmented equally, even in extremely unbalanced circumstances, a queuing order still cannot support efficient mechanisms for synchronization.

The idea of fine-grained synchronization operations include little data or code, and minimize serialized processing for maximizing parallelism. However, different communication patterns, applications, unbalanced workloads, and irregular communication algorithms are inevitable, and may cause large overheads. These problems degrade parallel performance and potential scalability. The current proposed methods may not suffice in applications for a large number of processors. Particularly, as the number of processors increases, these problems become more serious.

This study proposed a flexible mechanism that can configure the relationship of dependence among parallel and develop the synchronization without incurring networks in the waiting time period. Most studies have proposed numerous solutions to reduce network conflict for synchronization. In addition to this point, the chief concern of this study is the improvement regarding dependent relationships among parallel threads [7]. Because an efficient synchronization does not only comprise a reduction of network contention, the relationship such as different communication patterns, applications, unbalanced workloads, and irregular communication algorithms, is another significant factor. However, implementing these techniques simultaneously effectuates another problem, which indicates how to define the relationship in multiprocessor system. To

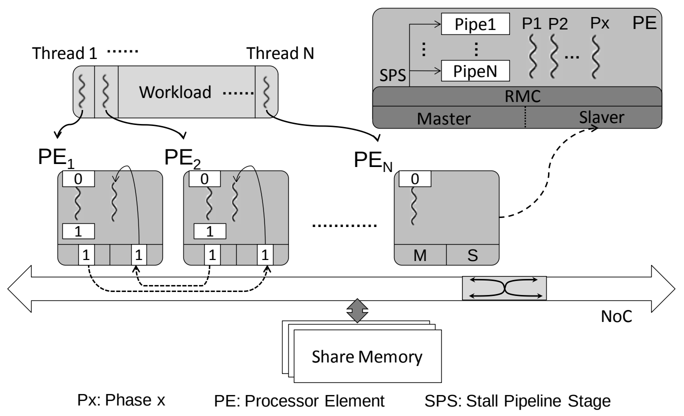


Fig. 1. Multi-core system with the proposed rendvzous mechanism.

overcome the issue, this study designed a novel architecture that can perform a rendezvous mechanism by load-store instructions directly and define the dependence in the local processing element. The experimental result demonstrates that the proposed mechanism and design can provide the efficient synchronization.

The remainder of this paper is organized as follows: Section II presents the synchronization and communication among multiple cores; Section III describes the design and proposed rendezvous mechanism; sections IV and V show the simulation experiment and simulation results, respectively; and finally, Section VI offers a conclusion.

II. Related work

A variety of hardware techniques have proposed for network contention. In the previous research [29], the rendezvous mechanism is implemented at the AHB, called RAHB. The RAHB interface added the slave organizations in each core's AHB interface to snoop the data on a bus passively. If data are required in the allocation, this node can read the current data on a bus directly. However, this mechanism only presenting the solution for data communication is insufficient.

Considering optimization of the synchronization mechanism, many literatures proposed numerous solutions. Mellor et al. presented a list-based queuing lock, First-in-first-out behavior, for improving network contention [29]. In [22], the authors developed the queuing lock to implement the management of spinlocks and backoff with no contention in a special period. Goodman et al. also performed locking primitives with a queue-based lock [24]. The study by [20] presented a synchronization-operation buffer based on the local management of a spinlock and events. From the viewpoint of synchronization, this buffer algorithm provided the policy of first-in-first-out order to manage synchronization in the local memory controller. These approaches developed synchronization mechanisms for barriers by the popular and simple queuing lock. However, the unbalanced workload cannot follow the FIFO order smoothly, and may incur a large number of unnecessary network contentions. Unfortunately, no workload can guarantee that the arriving time of every thread is closed. Even certain programs have extremely unbalanced arrival times [27]. The centralized barrier, which must tolerate the worst-case thread, cannot provide the efficient synchronization.

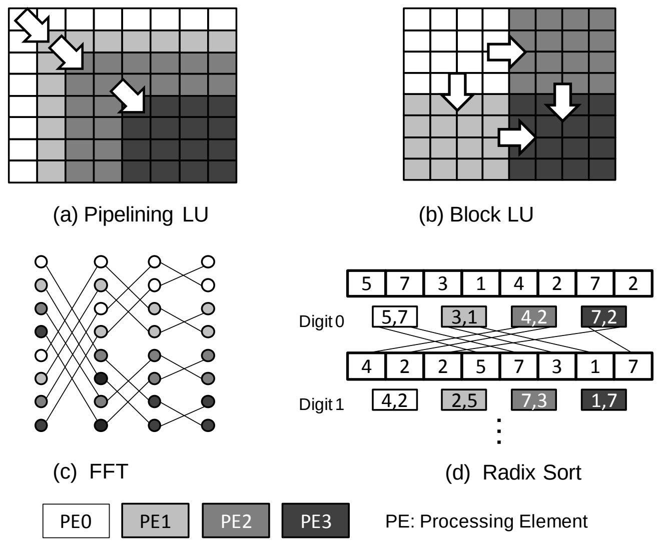


Fig. 2. Modeling producer-consumer communication

The study by [29] evaluated these different communication patterns for multi-core systems from the viewpoint of the producer-consumer. They classified the communication patterns based on the producer-consumer relationship. These communication patterns are shown as fig. 2. Fig. 2(a) and (b) show the example of one-to-one and one-to-some communication patterns. In the one-to-one communication pattern, the same row and column can be used as a parallel thread. This workload is separated to many parallel threads along with the row and column. After a producer (PE0) calculation, which proceeds to latter one sequentially, another PE of corresponding with the next row and column will obtain result. The fig. 2(b) is Block-LU based on the LU decomposition from SPLASH2 benchmark [26]. After a producer (PE0) produces results, it is communicated along the row and column consumers (PE1 and PE2). In this paper, the all-to-all communication pattern is also divided into two cases. The fig. 2(c) is the regular all-to-all communication pattern because the time to operate butterfly operations is fixed. In the fig. 2(d), because these sorted data are irregular arrangements, the time for when one PE finishes a thread is unpredictable. The radix sort is called an irregular communication pattern.

In [29], this paper demonstrated the significance of the producer-consumer relationship for multi-core system, with each core producing or consuming the results to other cores in the synchronization procedure. The traditional mechanisms using queuing behavior (First-in-first-out order) do not suffice for handling a variety of communications. To develop an efficient and scalable mechanism, the proposed design develops a flexible mechanism that can configure the dependent relationships among parallel workloads.

III. Rendezvous Mechanism

In this paper, the proposed rendezvous mechanism, which enhances the synchronization in local processing elements (PE),

```
//This function initials related flags and synchronous sequence.
Initial_relationship (related_flag *DR, related_flag *SR){    //DR:Destination Relative array, SR:Source Relative array,

   bool  DR[the number of cores];                    //Define the data structure for related_flag.
   bool  SR[the number of cores];                    //The index of array, which exists in the local element, signifies the corresponding
                                                     // positions core.
                                                     //The logic 1 in Relative array indicates that the local element has a relation of
                                                     //dependence with the corresponding positions cores.
   for each index in the DR and SR array {
     DR[index] = the flag of relationship for the destination    //Define the relationship based on the dependence of destination.
     SR[index] = the flag of relationship for the source         //Define the relationship based on the dependence of source.
    Synchronous_sequence[the number of cores]=0;                 //Initial all the synchronous sequences.
   }end for
}
```

Fig. 3. The Initial_relationship function to the initial parameter

```
//This function checks whether all destinations of related threads have been completed based on indexes corresponding with positions cores.
//The synchronous sequence indicates the order of the current stage in the other related cores.
//The returned value is used to determine whether the local element is allowed to proceed onto the next stage.
Check_ related_flag_for_destination (related_flag *DR){
    bool  DNS_flag=1;                              //DNS_flag: Determent Next Stage flag;
                                                   //DNS_ flag determines whether the local element can proceed to the next stage.
                                                   //If at least one synchronous_sequence is invalid, the DNS_flag becomes 0.
    for each index in the DR array {               //Within related threads, check all synchronous sequences corresponding with
        if  (DR[index]==”1”){                      //the index of array.
            if(Synchronous_sequence[Local] <= Synchronous_sequence[index]+n)   //That the condition is valid indicates that this local
                DNS_flag = DNS_flag & 1;                                        //element is allowed to proceed onto the next stage.
            else
                DNS_flag = 0;                      //One of all related conditions is invalid. The local element is stalled to
        } end if                                   // maintain the correct sequence among parallel threads.
      else
        DNS_flag = DNS_flag;                       //In the case of unrelated threads, the DNS_flag is preserved.
      }end for
      return DNS_flag                              //Finally, return the DNS_flag. If DNS_flag is log 1, the local element can proceed
  }                                                // onto the next stage.

//The input of this function and the conditional expression is different with Check_ related_flag_for_destination, as in the following
description.
// The returned value indicates that the local element is out of order. Therefore, it can be used to determine whether the local element must be
stalled to maintain the correct sequence.
Check_ related_flag_for_source (related_flag *SR){
  ………………
        if(Synchronous_sequence[Local]<=Synchronous_sequence[index])
  ………………
  }
```

Fig. 4. The algorithms of the checking function

reduces the collision among PEs and improves the flexibility of communication by defining relationship. The basic idea is to manage synchronization based on the dependent relationship in the local PE. In this way, the proposed mechanism can avoid the polling to reduce the contention and decrease the waiting time due to the unbalance threads. In the remainder of this section, we separately describe how the rendezvous mechanism works without polling in the subsection A and how this flexible communication achieves the efficient synchronization in the subsection B. Finally, we will compare the traditional and rendezvous mechanism based on dependent relationship in detail.

A. The communication by the Rendezvous mechanism

As shown in Fig. 1, the interconnection interfaces of PEs include the master and slave devices. In only the master interface, the flexibility of communication is constrained because every core can only passively obtain synchronous flags from other cores. That is, polling is the consequence of the architecture. The proposed rendezvous mechanism is shown in Fig. 1, and adds the slave port and the Rendezvous Mechanism Controller (RMC) between the PEs and interconnection. Not only every PE can transfer flags actively by master but obtain flags passively by slave. In traditional design, when one PE must obtain flags from others in the waiting period, it will require continuous polling of a shared variable. In rendezvous mechanism design, the waiting PE can be held by the RMC to obtain synchronous flags from the others without polling. When the PE of dependent relationship transfers flags by broadcast, the slave interface of the waiting PEs will can snoop the

```
//This function is the main operation. After defining all the relationships, the operation runs until all the stages are complete.
Operation_proposed_mechanism(related_flag *DR, related_flag *SR){
  Initial_relationship (related_flag *DR, related_flag *SR);          //Reset all synchronous_sequence and define SR and DR.
  do{
     if (Check_ related_flag_for_destination (related_flag *DR) && Check_ related_flag_for_source (related_flag *SR)){
                                                                  //If two conditions are valid, the local element is allowed to
                                                                  //run the next stage thread
     Run the next stage thread;
     Local_Synchronous_sequence++;                                //update local Synchronous_sequence after completing the current stage.
     Send_flag(local_Synchronous_sequence, local_element_number);
                                                                  //Inform local_Synchronous_sequence and local_element_number to other
                                                                  //cores by broadcast.
     } end if
     Else
       Receive_flag();

  }While (!(all stages are completed))
}
```

Fig. 5. The operation of proposed mechanism based on the dependence of relationship.

corresponding flags. To complete this mechanism, each of PEs must record the identity of the all PEs which have the dependent relationship. If the corresponding PEs broadcast a new flag, the slave interfaces of PEs which have the dependent relationship will update the new flag in the local PE.

The rendezvous communication, as shown in Fig. 1, has the master and slave register bank map, which are mapped on the unique memory address that correspond with the register bank of PE. The value of the white box indicates the execution order. The dependent relationships among parallel threads are configured in the RMC. As shown in PE1 of Fig. 1, the synchronization flag is broadcasted from the master of PE1 to the slave of PE2. Because the RMC supports the master and slave port, every core can inform actively and received passively by others. In the waiting period, the RMC replaces the polling mechanisms or interrupts handlers. After completing one thread of pahse1, PE2 checks the synchronous sequence without incurring networks by receiving passively. If PE1 completes the thread of the current phase, PE2 executes the next phase directly without stalling. This defined relationship based on applications offers increased flexibility in communication. The overhead of the conflicts on interconnection and redundant transfer for synchronization can be improved to the constant number of network contention.

B. The Rendezvous mechanism for the dependent relationship

The relationship of data dependence is defined to two relationships. *Source relation*: One PE must obtain data from the other PEs. This relation between one PE and the other PEs of source is called *source relation*. *Destination relation*: One PE must send data to the other PEs. This relation between one PE and the other PEs of destination is called *destination relation*. This rendezvous mechanism consists of two main operations, initial and checking relationship functions, as shown in Figs. 3 and 4. In the initial functions, the *source* and *destination relation* must be defined in *Destination Relative array* (DR) and *Source Relative array* (SR) based on the application beforehand. For example, PE_2 in Fig. 2(a) is the destination node of PE_1, and PE_1 is the source node of PE_2. Therefore, the index of the DR corresponds with the PE position. The content of DR records the relation of destination. If the current node (PE_1) has the relation of destination with PE_2, as shown in Fig. 2(a), the DR [index(2)] in the local memory of PE_1 records 1. Similarly, the SR (Source Relative array) is used to record the relation of the source with the local element. In the case of Fig. 2(a), the SR [index(1)] in the local memory of PE_2 is set to the logic 1. Another variable, synchronous sequence, indicates the number of completed stages corresponding with processing element positions, signifying the order of threads among processing elements. When a local element completes the current stage, this synchronous sequence in the local element accumulates and broadcasts to inform others which have the dependent relationship. In other words, this value can be used to retain the correct order among threads. The synchronous sequence must be monitored by RMC to guarantee that the synchronous sequence of the local element does not precede a stage for dependent PEs.

The checking relationship function is used to maintain the correct order among threads. Therefore, every flag in DR and SR should be checked to guarantee that the current stage is valid. The *Check_related_flag_for_destination* and *Check_related_flag_for_source* function check all relative flags as shown in Fig. 5. If all the sequences are valid, the *DNS_flag* (Determent Next Stage flag) is the Logic 1. If the *DNS_flag* of source and destination is true, the local element is allowed to proceed onto the next stage. When this element completes a current thread, the synchronous sequence accumulates and updates other cores by broadcast. Conversely, if the determination is false, the local element passively waits for new synchronous sequences. Because the slave interface exists in every local node, the waiting node can obtain a new synchronous sequence passively. In the waiting period, this does not cause unnecessary access for a network. The line 5 of Fig. 4 indicates that the PEs of the data dependence can allow ahead of *n* stages for the local PE. Therefore, this mechanism is not limited by the worst-case thread within *n* stages.

The proposed mechanism develops two main operations. First: when the one PE conforms to the dependent relationship,

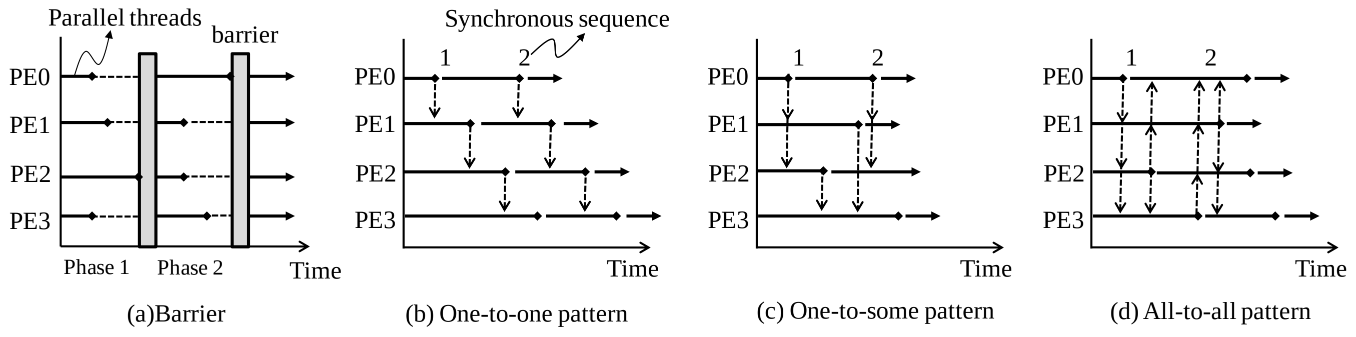


Fig. 6. Traditional and Rendezvous mechanism for communication patterns

the next thread can be handled immediately without waiting the worst-case thread. Second: owning to data dependence, one PE must wait the others. the proposed mechanism allows this waiting PE to execute the next thread beforehand.

C. *Rendezvous mechanism based on communication patterns*

The proposed mechanism can define the data dependence among multi threads based on communication patterns. According to dependent relationships, communication patterns are classified to three cases: One-to-one, One-to-some, and all-to-all. The Pipelining-LU, Blok-LU, and radix-sort algorithms in Figs. 2(a), 2(b), and 2(c) are for one-to-one, one-to-some, and all-to-all communications, respectively. Figs. 3(b), 3(c), and 3(d) show examples of one-to-one, one-to-some, and all-to-all communication with the rendezvous mechanism, respectively [7][11]. The dot and slash blocks indicate that a PE is utilizing the bus for synchronization and using these load/store instructions to execute a program. Hence, lower-priority PEs cannot use shared resources when a higher priority PE_i occupies the shared interconnection. This network conflict is denoted by a dark block. The white block denotes that PE_i must wait for other PEs before continuing to the next phase. The gray block indicates that a PE is running a program without requesting bus access. In the gray block, the numeral indicates the number of completed phases in the local PE. Finally, the direction of the arrow indicates a PE with a completed current phase that informs to other relative PEs. Beside, the value near the arrow indicates that a synchronous sequence is transferred from the PE of a completed current thread to relative PEs based on the application. The synchronous sequence indicates the order of threads corresponding to other related PEs. For All-to-All communication pattern, if every PE has the same current phase value, this means that all PEs are executing the same application. The remainder of this subsection details every communication shown in Fig. 6, as follows.

1) *Conventional mechanism*

Fig. 6(a) shows an example using the conventional mechanism. In the centralized barrier, all PEs must wait the worst-case PE until this PE has completed the computations for the current thread, and transferred the flags to the others (PE_1, PE_2 and PE_3). On the other hands, these waiting PEs will incur continuous polling in the waiting period. To wait lowest threads and incur excessive access of interconnection, the conventional mechanism cannot develop the efficient communication for synchronization.

2) *Rendezvous mechanism*

The dependence relationship for pipelining-LU, as shown in Fig. 2(a), is a one-to-one example of four PEs managing LU-decomposition simultaneously. Fig. 6(b) shows a one-to-one communication pattern with the proposed rendezvous mechanism. When PE_2 has completed the current thread, it forces the network to transfer the synchronous sequence to PE_3, which subsequently operates the next row and column. When PE_2 waits for the synchronous sequence from PE_1, PE_2 does not access the network for synchronization management. Until PE_1 sends the synchronous sequence to PE_2, the next phase in PE_2 can be operated.

Fig. 6(c) shows an example of one-to-some communication patterns. As the above-mentioned relation, the PE_0 has destination relation with PE_1 and PE_2. After PE_0 completes the current thread, the synchronous sequence will be accumulated to 4 and informed to PE_1 and PE_2. Similarly, when the PE_1 and PE_2 complete the current threads, the accumulated synchronous sequences are sent to the destination node (PE_3).

In the all-to-all communication pattern, Every PE that has completed the current thread must wait for others. Each other PEs have destination and source relations simultaneously. In this paper, the all-to-all communication pattern will be classified to irregular and regular communication based on applications. The FFT consists of butterfly operation. Every butterfly is parallel operation and has the same computation. Because the irregular data arrangement causes the different computation in each of PEs, the time for when one PE finishes a thread is unpredictable. The radix sort is irregular all-to-all communication patterns [7][8][9]. The proposed mechanism allows PEs, which have completed the current threads, can proceed to the next thread. As Fig. 6(d), when PE_0 completes the current thread and sends the synchronous sequence to others, this PE will execute the next thread continuously.

In summation, the proposed mechanism develops two main flexible operations. First: When the one PE conforms to the dependent relationship, the next thread can be handled immediately without checking all PEs. Second: when one PE must wait the others, which have the dependent relationship, the

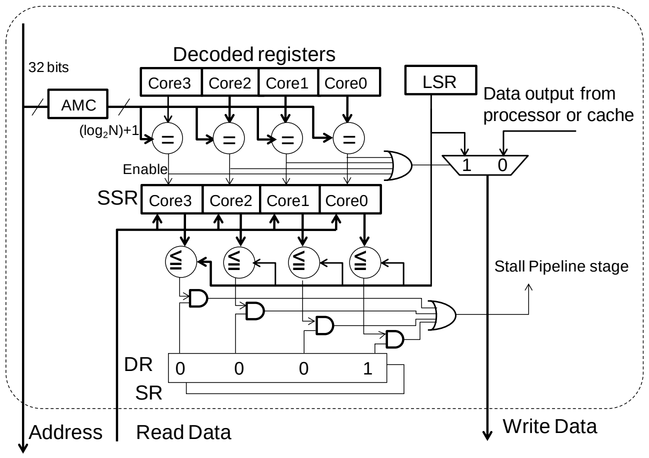


Fig. 7. Hardware organization to check synchronization

proposed mechanism allows that this PE, which should wait, can execute the next thread beforehand.

D. *Hardware organization*

This section describes the implementation of the rendezvous mechanism. Fig. 7 shows that the RMC minimizes the incurring network contention for synchronization. Because every PE has a unique mapping address to identify the source of synchronous sequence, the indices of SR and DR can indicate where the synchronous sequence are derived from. The synchronous sequences are stored in the SSR (Synchronous Sequence Register), which is 8*N bits d flip-flop, retaining the synchronous sequence of all PEs to decide between run and hold threads of the local PE. Moreover, the Local Synchronization Register (LSR) reserves the synchronous sequence of the local PE and compares it with a relative synchronous sequence. The AMC (Address Match Controller) identifies whether the addresses are mapped to a position of a processing element.

In the design, the decoded registers are compared with the AMC. If one of the registers corresponds with the AMC, this means that the address is derived from a processing element, and the Read Data indicates the corresponding synchronous sequences with the processing element. If at least one of the synchronous sequences that have dependence relationships is invalid, the *stall pipeline stage* signal stalls the pipeline of the local element. This design can hold and start the local element immediately rather than use the loop to lock the current state. Furthermore, in the waiting period, the design obtains the synchronous sequences passively without incurring a network.

IV. Simulated Experiment

This section provides an overview of the experimental environment. In addition, this study classifies three communication patterns based on producer-consumer relationships, and introduces these patterns in Section IV.B.

TABLE I. Environment of simulation

Simulated environment	*Parameters of the target architecture*
Multicore Architecture	Symmetric Multiprocessing (SMP)
Simulated processors	ARM 9
Number of processors	4~16
Processors clock frequency	200 MHz
Interconnect clock frequency	100 MHz
Interconnect Protocol	AHB
Interconnect data size	1 Bytes
Share memory space	512 Kbytes
Share memory page size	4 Kbytes

A. *Simulated Framework*

Table I displays the simulation environment. This study compared the traditional and proposed synchronization mechanism in Table I of the simulated environment. The C code of a parallel program was compiled into the ARM9 instruction set and placed in the instruction memory of each processor. The data bus architecture is for AMBA [3][12-17]. The RMC is modeled as the communication interface between the processor and AHB. If two or more cores require the bus, the bus arbiter grants one of the requiring cores with priority. According to the AMBA protocol, a basic transfer comprises an address and data phase, which cost a minimum of two cycles. Because this study focused on multiprocessor communication, the access of other devices over a bus, such as DMA, was not considered. Bus access latency is assumed to cost two cycles [3][23][28].

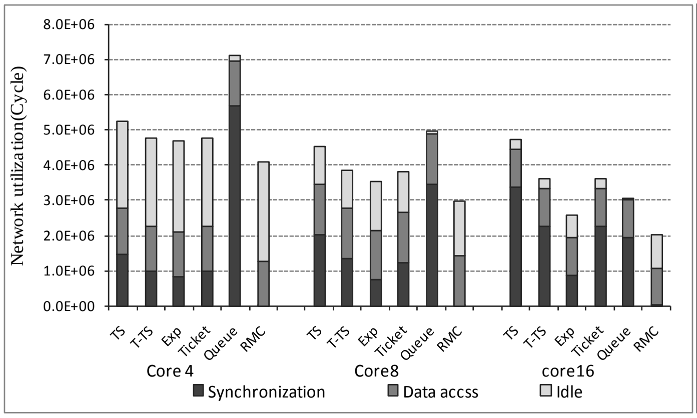
B. *Communication pattern for multi-core architecture*

This study classifies the dependence relationship based on producer-consumer communication, and selects applications from the benchmarks in Table II, as follows: one-to-some (Block-LU); one-to-one (Pipeline-LU); many-to-many (FFT and Split Radix sort) [8][25][26][29].

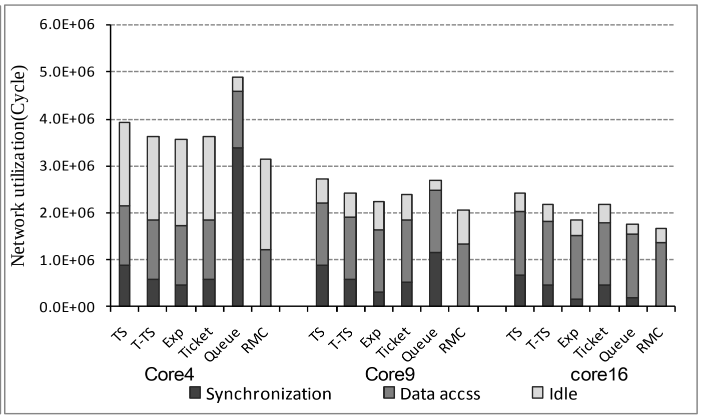
In this one-to-one communication pattern, all the elements of the row and column depend on the previous row and column. Each processor is required to synchronize with another processor that executes the previous row and column. In this Block-LU case, a two-dimensional matrix is divided into blocks, which can be separated into several parallel threads that can be executed by multiple cores. Fig. 2(b) shows a case in which four divided blocks are executed by four processors. Finally, the split radix sort is a parallel version of the standard radix sort [7][8][9]. The N elements can be divided to P processors. Every processor handles N/P elements simultaneously. Fig. 2(c) and 2(d) are examples of implementation that 4 processors operate split

TABLE II
The selected application of benchmarks

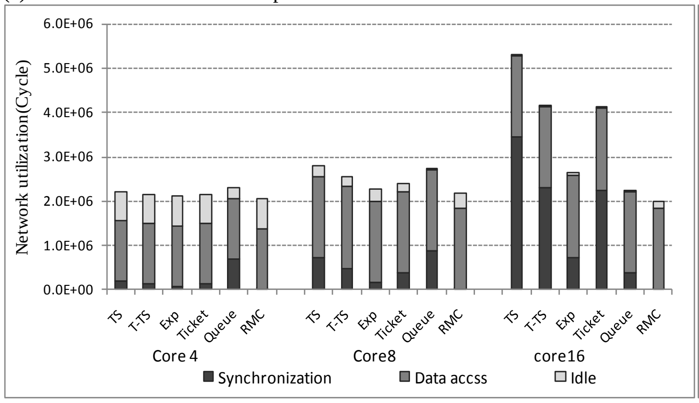
	Communication pattern	*Description*
FFT	All-to-All irregular	8-k complex data points (All-to-All irregular communication)
Block-LU	One-to-Some	128*128 Lower/upper LU-decomposition
Pipeline-LU	One-to-one	LU-decomposition for pipelining schedule
Radix Sort	All-to-All regular	16-k element integer radix sort



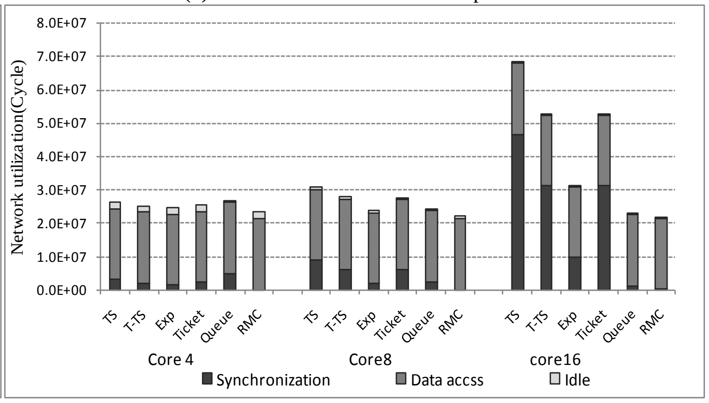
(a)One-to-One communication pattern



(b) One-to-Some communication pattern



(c) All-to-All irregular communication pattern



(d) All-to-All regular communication pattern

Fig. 8. Simulation for communication patterns with benchmark

radix sort for 8 random 3-bit elements in share memory and 8 points FFT.

The following section shows the simulations of these conventional five proposed synchronization mechanisms (test-&-set, test-test-&-set, exponential test-&-set, ticket lock, queuing behavior, and RMC) and communication patterns. Furthermore, one-to-some and some-to-one are similar communications in the multiprocessor architecture. This study evaluated one-to-some and some-to-one with the Block-LU.

V. Performance

The network utilization of all communication patterns is shown in Fig. 8 for 4~16 numbers of core systems. The idle indicates that the period has no requests to incur the network because all the PEs were executing threads or waiting without incurring access for load/store instructions or synchronization. In other words, the bus is free during this period. If any cores require accessing, the dark block can be used, which indicates the synchronization, and the light block indicates the data access. Furthermore, TS, T-TS, and EXP are abbreviated from Test & Set, Test-Test & Set, and Exponent backoff, respectively.

Fig. 8(a) shows a one-to-one pattern for LU decomposition. The workload of LU decomposition is not balanced because these elements of a matrix at the bottom right have a more complex operation and more access than that shown in the upper left corner of the array. Therefore, the queuing behavior, followed by the FIFO order, incurs a large number of networks for synchronization. As the number of cores increases, polling occurs for the spinlock continuously, growing significantly. Even TS, T-TS, EXP, and the ticket cause more network utilization than the queuing order. The RMC accesses the network only when sending synchronous sequences. Because the dependence is series for one-to-one pattern, the restrictions of parallelism are severer than other patterns. As the number of PEs increases, the proposed RMC has a large proportion in the idle time and the constant number of access for synchronization in this pattern.

As Figs. 4(b), shown The RMC can utilize the network in the parallel benchmark efficiently due to the flexibility for defining the relationship. A comparison between Figs. 4(a) and 4(b) shows that the ratio of idle cycles in Fig. 8(a) is higher than that of the one-to-some communication for the RMC because the serious communication, not the parallel pattern, results in more waiting time in every PE. By increasing parallelism, the opportunity of valid communication grows among PEs. Therefore, the idle time in the RMC can retain a lower ratio than the one-to-one pattern as the parallelism and the number of cores increase.

TABLE III
RMC VERSUS OTHER MECHANISMS FOR SPEEDUP AND ROS

Benchmark	*n*	Test&Set		Test-Test&Set		Exponential Backoff		Ticket		Queuing		RMC	
		Speedup	ROS	Speedup	ROS	Speedup	ROS	Speedup	ROS	Speedup	ROS	Speedup	ROS
Block-LU	4	1.609	22.287	1.738	16.051	1.770	13.190	1.738	16.055	1.291	69.082	2.007	**0.096**
	9	2.323	32.619	2.606	24.401	2.796	13.622	2.648	22.363	2.340	42.726	3.052	**0.290**
	16	2.600	28.445	2.873	20.953	3.432	9.698	2.875	20.901	3.575	10.877	3.791	**0.722**
Pipeline-LU	4	1.130	27.683	1.245	20.332	1.258	17.182	1.245	20.332	0.833	80.110	1.448	**0.073**
	8	1.483	44.494	1.741	34.830	1.895	20.461	1.769	32.190	1.353	69.310	2.270	**0.202**
	16	1.070	71.577	1.405	62.682	1.976	33.642	1.406	62.613	1.658	63.801	2.485	**0.588**
Radix-sort	4	2.180	8.660	2.245	5.945	2.267	2.838	2.245	5.942	2.091	29.649	2.330	**0.102**
	8	1.728	25.996	1.892	18.975	2.131	7.249	2.014	16.232	1.762	31.982	2.213	**0.193**
	16	1.037	64.743	1.322	55.044	2.086	27.613	1.340	54.648	2.474	17.097	2.784	**0.424**
FFT	4	1.214	12.236	1.266	8.505	1.298	6.362	1.255	8.691	1.213	19.125	1.370	**0.167**
	8	1.036	29.051	1.147	21.445	1.334	7.785	1.149	21.424	1.340	10.668	1.434	**0.350**
	16	0.469	68.723	0.609	59.433	1.020	31.036	0.609	59.403	1.418	5.470	1.455	**0.711**

Fig. 8(c) and 4(d) show the all-to-all patterns. Compared with the FFT application in Fig. 8(c), the radix-sort is an irregular communication pattern. The results of the simulation show that the queuing behavior in the irregular pattern must spend more network utilization for synchronization compared to the regular pattern. Other conventional mechanisms also incur serious network contention. According to the simulation, the proposed mechanism reduces contention and bandwidth consumption in regular and irregular all-to-all cases. The RMC can maintain lower network contention, even when the multi-core system has more cores or must handle complex applications.

Table III shows the Ratio of Synchronization time (RoS) and speedup based on single-core performance. The ROS denotes the proportion of synchronization occupancy to total network utilization. In these applications, in the environment of higher parallel degrees, workloads with TS, T-TS, EXP, and ticket mechanisms cannot achieve efficient improvement. In other words, these applications produce heavy network traffic with an increase in the number of PEs, which results in a performance bottleneck in multi-core systems. Although the queue-based mechanism improves throughput as the number of PEs increases, the irregular all-to-all communication (radix-sort) and the unbalanced workload (pipeline-LU) incur such an increase in bus utilization that performance cannot be improved efficiently.

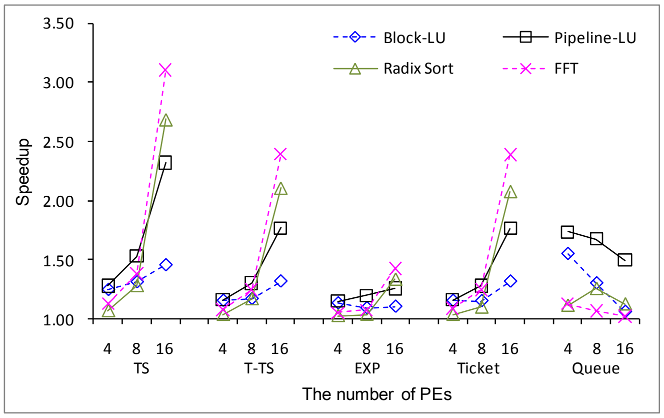


Fig. 9. RMC Speedup with other mechanisms.

Therefore, in most cases, the RMC can sustain ROS at less than 1 % to obtain an enhanced throughput.

Fig. 9 shows the comparison of the RMC with other mechanisms. An increase in the number of cores corresponds with a dramatic decrease in throughput for TS, T-TS, and ticket mechanisms. Although the exponential backoff has superior performance, the problem of a prolonged time slot still cannot be solved. Considering the scalability, the queuing behavior in irregular communication is unsuitable in a large number of cores.

Communication patterns, the number of cores, unbalanced workload, and irregular communication are significant factors for impacting network contention for synchronization. Due to heavy network utilization, these factors limit the scalability and performance of the multi-core systems critically. Because the RMC only requires incurring networks when threads are complete, it eliminates the redundant synchronization contention. Furthermore, this design for communication stalls the pipeline stage directly without polling, which reduces the latency to handle the synchronization production. As shown by the simulation results, the four impact factors of incurring network contention (communication patterns, the number of cores, unbalanced workload, and irregular communication) can improve the scalability and performance of multi-core systems.

VI. Conclusions

Efficient synchronization is essential for high performance in workloads of parallel threads. Multithreads on multi-core systems generate serious communication overhead for synchronization. This paper discusses network utilization based on communication patterns, thread dependence, irregular or regular synchronization, and the number of PEs. The proposed mechanism not only achieves low overhead synchronization, but also improves thread management by flexible communication based on the dependent relationship for applications. Furthermore, the simulation results show that RMC reduces network utilization significantly, and improves speedup in the benchmark factors that impact multi-core systems.

REFERENCES

[1] P. P. Gelsinger, “Microprocessors for the New Millennium: Challenges, Opportunities, and New Frontiers” in Proc. Int. Conf. Solid-State Circuits Conference, Feb. 2001, pp. 22-25.

[2] Y. Like, S. Qingsong, C. Tianzhou and C. Guobing, “An On-chip Communication Mechanism Design in the Embedded Heterogeneous Multi-core Architecture”, in Proc. IEEE Int. Conf. on Networking, Sensing and Control, Apr. 2008, pp.1842 – 1845

[3] ADDISON-WESLEY, “ARM system-on-chip architecture second edition”

[4] J. L. Hennessy and D. A. Patterson.: ‘Computer Architecture A Quantitative Approach’ (Morgan Kaufmann Publichsers,2003,3rd).

[5] Z. Fanga, L. Zhangb, J. B. Cartera, L. Chenga, and M. Parkerc, “Fast synchronization on shared-memory multiprocessors: An architectural approach”, Journal of parallel distributed computing, Apr. 2005, pp.1128-1170

[6] G. Narayanaswamy, P. Balaji and W.Feng, “Impact of Network Sharing in Multi-Core Architectures ” Digital Object Identifier, Aug. 2008 pp:1-6

[7] Byrd, G. T., and Flynn M. J.,“Producer–Consumer Communication in Distributed Shared Memory Multiprocessors” Proceedings of the IEEE, Vol. 87, NO. 3, Mar, 1999, pp.456-466

[8] G.E. Blelloch, C.E. Leiserson, B.M. Maggs, C.G. Plaxton, S.J. Smith, and M. Zagha “A Comparison of Sorting Algorithms for the Connection Machine CM-2.” in Proc. Int. ACM Symp. on Parallel Algorithms and Architectures, Jul 1991, pp. 3-16,

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. "Introduction to Algorihms." The MIT Press and McGraw-Hill, 1990.

[10] S. Pandey, M. Glesner, and M. M.. uhlh.. auser: “ Performance aware on-chip communication synthesis and optimization for shared multi-bus based architecture.” in Proc. Int. 18th ACM Symp. on Integrated circuits and system design, Sep 2005, pp. 230-235

[11] Guy E. Blelloch, “Vector Models for Data-Parallel Computing”, The MIT Press Cambridge, Massachusetts London, England

[12] Synopsys, “DesignWare AHB Verification IP Databook,” http://www.synopsys.com/products/designware

[13] P. V. Knudsen and J. Madsen, “Communication estimation for hardware/software codesign” in Proc. 6th international workshop on Hardware/software codesign. 1998, pp. 55–59

[14] P. Knudsen and J. Madsen, “Integrating communication protocol selection with partitioning in hardware/software codesign,” in Proc. Int. Symp. System Level Synthesis, Dec. 1998, pp. 111–116.

[15] Guo, Jianjun; Dai, Kui; Lai, Ming-Che; Wang, Zhiying” The P2P Communication Model for a Local Memory based Multi-core Processor” in Proc. IEEE Int. Conf. on Young Computer Scientists, Aug. 2008, pp:1354 – 1359

[16] X. Zhu and S. Malik, “A hierarchical modeling framework for on-chip communication architectures,” in Proc. Int. Conf. Computer Aided Design, Nov. 2002, pp. 663–671.

[17] S.R. Alam, R.F. Barrett, J.A. Kuehn, P.C. Roth, and J.S. Vetter, " Characterization of Scientific Workloads on Systems with Multi-Core Processors,” in Proc. Int. IEEE Symp. on Workload Characterization, Oct. 2006, pp.225-236

[18] L. Lamport. “A new solution of Dijkstra’s concurrent programming problem.” Communications of the ACM, pp. 453-455, Aug. 1974

[19] A. Silberschatz, P. B. Galvin, G. Gange, “Operating Systems Concepts with Java,” John Wiley and Sons. 7Ed 2005.

[20] M. Monchiero, G. Palermo, C. Silvano and Villa, O, “Efficient Synchronization for Embedded On-Chip Multiprocessors,” IEEE Trans. on Very Large Scale Integration (VLSI) Systems, vol. 14, no. 10, pp. 1049-1062, Oct. 2006.

[21] V. S. Pai, P. Ranganathan, and S. V. Adve “The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodology” in Proc. Int. ACM Symp. on High Performance Computer Architecture, Feb. 1997, pp.72-83

[22] T. Anderson, “The performance of spin lock alternatives for shared memory multiprocessors,” IEEE Trans. Parallel Distrib. Syst., vol. 1, no. 1, pp. 6–16, Jan. 1990.

[23] P. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, and G. Nicolescu, “Parallel programming models for a multi-processor SoC platform applied to high-speed traffic management,” in Proc. Int. Conf. Hardware/software codesign and system synthesis, Sep. 2004, pp. 48-53.

[24] A. Kagi, D. Burger, and J. R. Goodman, “Efficient synchronization: Let them eat QOLB,” in Proc. 24th Ann. Int. Symp. Comput. Arch. (1SCA), 1997, pp. 170–180.

[25] J. Baxter, “Stanford Parallel Applications for Shared Memory (SPLASH),” 2001 [Online]. Available: http://www-flash.stanford. edu/apps/SPLASH/

[26] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta.”The SPLASH-2 Programs: Characterization and Methodological Considerations,” in Proc. 22th Ann. Int. Symp. Comput. Arch. 1SCA, Jun. 1995, pp.24–36

[27] W. Qin and S. Malik, “Flexible and formal modeling of microprocessors with application to retargetable simulation,” in Proc. Design, Automation and Test (DATE), Mar 2003, pp.556-561.

[28] ARM926EJ-S Performance, Power & Area [Online]. Available: http://www.arm.com/products/processors/classic/arm9/arm926.php

[29] G. T. Byrd and M. J. Flynn, "Producer-consumer communication in distributed shared memory multiprocessors", Proc. IEEE, vol. 87, pp.456 - 466 , 1999.

[30] J. M. Mellor-Crummey and M. L. Scott, “Algorithms for scalable synchronization on shared-memory multiprocessors,” ACM Trans. Comput. Syst., Vol. 9, No. 1 Jan. 1991, pp. 21–65,