System-on-a-Chip Processor Synchronization Support in Hardware

Bilge E. Saglam
Georgia Institute of Technology
School of Electrical and Computer Engineering
Atlanta, GA 30332
bilge@ece.gatech.edu

Vincent J. Mooney III
Georgia Institute of Technology
School of Electrical and Computer Engineering
Atlanta, GA 30332
mooney@ece.gatech.edu

Abstract

For scalable-shared memory multiprocessor System-on-a-Chip implementations, synchronization overhead may cause catastrophic stalls in the system. Efficient improvements in the synchronization overhead in terms of latency, memory bandwidth, delay and scalability of the system involve a solution in hardware rather than in software. This paper presents a novel, efficient, small and very simple hardware unit that brings significant improvements in all of the above criteria: in an example, we reduce time spent for lock latency by a factor of 4.8, the worst-case execution of lock delay in a database application by a factor of more than 450. Furthermore, we developed a software architecture together with RTOS support to leverage our hardware mechanism. The worst-case simulation results of a client-server example on a four-processor system showed that our mechanism achieved an overall speedup of 27%.

1. Introduction

In a shared memory multiprocessor System-on-Chip (SoC), it is critical that two or more processors be able to execute on a common set of data structures or on some other shared piece of code (critical section), without hindering each other's work. For the processors to work properly and the shared data structure(s) to be consistent, support for synchronization is typically provided in the form of special instructions that guarantee an ordered, deterministic, i.e., atomic access to shared memory. In this paper, we focus on making the synchronization work in real-time by providing synchronization functionality in hardware.

Many current processors support instructions (special *load* and *store* instructions), which provide atomicity during read and write accesses to memory. With the functionality brought by these special instructions, traditional synchronization primitives have been developed in software, such as test-and-set, compare-and-swap, fetch-and-increment, fetch-and-add and many more. Using these primitives, several locking algorithms have been developed such as tournament locks [5], delay after noticing the release of a lock, delay between each reference (where delays may be static or may be set with exponential back off) and queuing in shared memory [1]. On the other hand, several cache-based locking primitives were developed and evaluated [2-4] as a hardware solution to the synchronization problem. These different approaches examine synchronization in terms of busy waiting of the processors, intrinsic latency for accesses to the synchronization variables in the memory and the network contention generated by these accesses. It is shown that a hardware solution brings a much better performance improvement [2] than the algorithmic locking alternatives developed in software. However, most of the hardware solutions introduced are nothing but improvements on processors caches in the form of private caches and introduction of new consistency models [6].

In this paper, we present a hardware mechanism that is capable of controlling the processor synchronization, thereby enabling us to dramatically reduce software overhead, improve performance criteria (such as latency, bandwidth consumption and delay) and totally eliminate the cache-coherency problems. The paper is organized as follows: Section 2 presents the background and motivation, Section 3 describes our methodology involving both software and hardware architecture designs, Section 4 presents our simulation environment and experimental setup, and finally Section 5 concludes the paper.

2. Background and motivation

Synchronization variables provide atomic access to shared memory locations through which multiple execution points (processes/threads/tasks) in an

application program can interact. An atomic locking allows only one task (that is holding the lock) to execute on a shared memory location or on a Critical Section (CS). In general purpose processors, special load-linked (`ll`) and store conditional (`sc`) instructions (e.g., 'LL' and 'SC' for MIPS4000 or 'lwarx' and 'stwcx.' in MPC860) are implemented in hardware. The `ll` and `sc` instructions are paired in such a way that both of them must reference the same physical address space (i.e., effective address –EA) in memory, otherwise execution of these instructions is undefined. Moreover, their execution establishes a breakable link between the two. The link between `ll` and the subsequent `sc` instruction will be broken if an external device has modified the value in the EA or an exception has occurred in the meanwhile (i.e, after `ll` but before `sc`). In this case, the store instruction fails to execute. If the link is not broken, the store instruction will succeed. In this way, the atomicity during accessing the EA in the memory is guaranteed [7].

These paired instructions are used to generate synchronization primitives (e.g., test-and-set, compare-and-swap, fetch-and-increment, fetch-and-add) which emulate a *lock* needed before entering the CS and thereby providing a higher level synchronization facility for the tasks. Therefore, using these primitives, the application program, with its multiple tasks that share memory, can be designed ensuring mutual exclusivity and consistency. For example, in the case of test-and-set, each processor checks the lock – tests whether the lock is free – first. If the lock is free, the processor acquires the lock by setting the lock variable. However, if the lock is busy (i.e., if the lock was previously set by another processor), then the processor must wait and try again later. In the latter case, the problem of *busy wait* arises; the processor will spin on executing test-and-set and will not be able to do other useful work until the lock holder releases the lock. Furthermore, the repeated test-and-set executions may degrade the communication bandwidth used among other processors, preventing them from doing other useful bus transactions and affecting their performance. Even worse, repeated test-and-set executions may cause an extra delay for the lock holder that wants to release the lock, because the lock holder also contends with the other spinning processors.

Definition 1: Lock Delay. Lock delay is the time between when a lock is released and when a spinning or otherwise waiting processor acquires the lock [1].

Example 1 Consider a web-server application program which consists of multiple client threads C_i (i=1,2…n) and server threads S_i . Let's say a client C_3 attempts to acquire a lock (which is currently held by S_1) in order to safely read from the shared memory space of the server. Clearly, C_3 fails to acquire the lock. When the server thread S_1 releases the lock, C_3 will be able to acquire the lock. The lock delay time spent by C_3 is the time between when S_1 releases the lock and when C_3 finally acquires the lock. □

Definition 2: Lock Latency. Lock latency is the time required for a processor to acquire a lock in the absence of contention [1].

Example 2 Again consider the same application in Example 1, but where the lock is available and there is only one client thread willing to acquire the lock. Then if the client attempts to acquire the lock, it will be successful. The lock latency is the time between when the client attempts to get the lock and when it acquires the lock. □

As a solution to the aforementioned problems, several software approaches provide more efficient spin–lock techniques for better performance. Spin-on-test-and-set, spin-on-read and the introduction of static or adaptive delay into the spin-wait loops (e.g., delay after noticing released lock or delay between references) are some of the most popular spin-lock alternatives [1]. However, these different methods are implemented in software and directly affect the execution flow of the software. Furthermore, the methods indicate poor performance behaviors in terms of bandwidth consumption, lock delay (Definition 1) and lock latency (Definition 2). Moreover, these different methods cause useful bus cycles to be wasted because of hold cycles (i.e., cache response time due to simultaneous cache invalidations in case of a lock release) [1-3]. Therefore, the efficiency of these techniques is dependent on the application program and the architecture, such as whether there are lots of processors making use of locks or how frequent the locking attempts occur in the application (i.e., how many CSes exist in the program). On the other hand, some previous work concentrates on cache schemes (e.g., weak consistency model) and associate private caches with each processor [2,6]. This model has to keep lock variables and the state information of these lock variables in caches and cache directory at each processor node, which is an overhead in the hardware design. Also, these hardware solutions are dependent on the memory hierarchy that supports a special consistency model [2].

We have devised a novel synchronization architecture as a solution to the processor synchronization problems when encountered in a System-on-a-Chip (SoC). Specifically, we propose moving some of the synchronization to hardware, which, in SoC design, can execute at the same clock speed as the processor itself. Furthermore, we ensure deterministic and much faster atomic accesses to lock variables via an effective, simple and small hardware unit. Our solution provides significant performance improvement in terms of lock latency, lock delay, bandwidth and scalability, making it suitable for real-time applications run on a multi-processor SoC.

3. Methodology

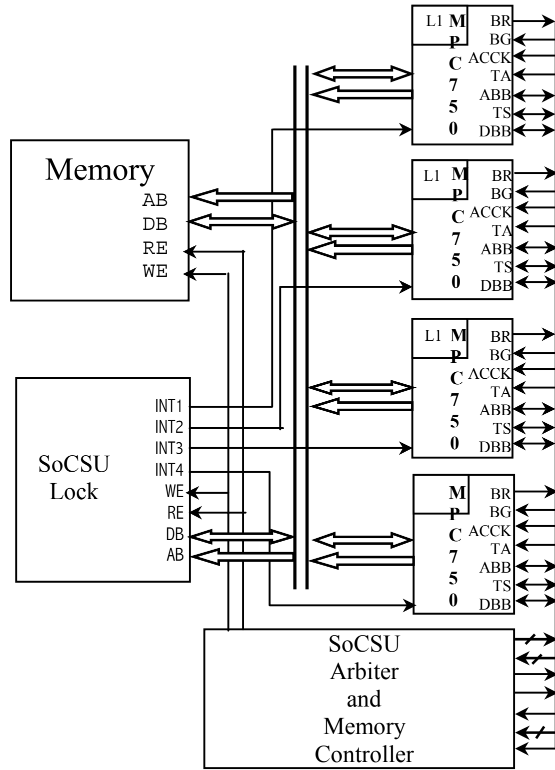

Figure 1. Simulation interface diagram for the hardware architecture.

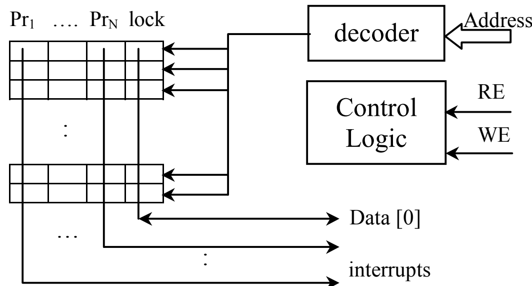

Figure 2. Basic SoCSU Lock architecture.

We implement the processor synchronization with a hardware mechanism which we call SoC Synchronization Unit (SoCSU). The SoCSU has a number of bit entries where each bit implements a single lock variable. For example, a SoCSU may have 256 such entries. The lock variable addresses are mapped into a common address range in every processor's address space. SoCSU is connected to the memory bus of each Processing Element (PE) through an arbiter/memory-controller that directs incoming access requests either to the memory or to the SoCSU (Figure 1).

The basic SoCSU Lock architecture for an N-processor SoC is shown in Figure 2. The architecture includes a set of N 1-bit Pr_i locations (where 'Pr_i' stands for 'Processor#i' and i ranges from 1 to N) associated with each lock variable. A boolean '1' in Pr_k indicates that PE_k has unsuccessfully tried to acquire the lock and so is waiting for the lock to be released. This boolean '1' is also used by the interrupt generation logic to send an interrupt to the waiting processor. When a lock is released, the associated Pr_i bits are checked in order to determine which processor is waiting for this lock so that an interrupt is sent to these waiting processors one at a time (in a priority or FIFO fashion).

Example 3 Consider that PE_2 attempts to acquire one of the locks, but fails. Then, the Pr_2 location for that lock entry will be set to '1' by the control logic. As soon as the lock holder releases the lock, an interrupt will be generated in the next clock cycle in order to notify PE_2. After this notification, the Pr_2 bit location will be cleared.□

SoCSU also includes a decoder unit which decodes the incoming address and enables the corresponding lock entry to start the transaction. The control logic block in Figure 2 handles writing of a '1' to the lock locations (for acquired locks) and interrupt generation, when a lock is released and other processors are waiting for the lock.

Before going into details about the hardware unit, we first examine how the software (both the C language level and the assembly level) will make use of this mechanism and what kind of instructions will start a transaction on the SoCSU.

3.1. Software implementation

Our mechanism provides lock access with a single instruction. The need for the special load (LL) and store (SC) instructions has been removed so that our mechanism can be applied to any general-purpose processor (whether the processor supports atomic load/store instructions or not). Moreover, the number of instruction cycles per CS is reduced. This directly results in the latency of the lock acquisition to be reduced from at least four instruction cycles to one instruction cycle.

As we can see in Figure 3 (b), the new assembly routine does not contain the special synchronization instructions LL and SC anymore. Rather, by the regular load instruction LW, the lock value from the lock variable address (which is the value stored in R1 below) is loaded into the target register R2, and the code leaves the rest of the test-and-set execution to the hardware. After getting the lock value into a temporary register R2, the program either jumps to 'sleep' or acquires the lock and gets into the CS. Here, just reading the lock value and seeing that it is a zero, implies that the lock is acquired automatically; i.e., there is no need to store a '1' back to that lock

address because the SoCSU Lock hardware does so automatically (SoCSU guarantees the atomic acquisition of the lock). On the other hand, if the semaphore address contained a ‘1’, this would mean that the processor cannot begin to execute the CS, and the processor must wait for the lock to be released, i.e., until an interrupt occurs to notify the processor. After the interrupt is sent to the processor, the program will get out of the infinite loop, jump to the external interrupt vector, and the Pr bit position will be cleared in the next access request to the SoCSU. For spin-lock, on the other hand, as shown in Figure 3(a), the lock acquisition consists of trials with two spinning loops (notice the bold-faced ‘**try**’ labels in Figure 3(a)). These loops are busy-wait loops that waste memory bus cycles. Figure 4 shows the significance of our approach in terms of these busy-wait loops. In case there is contention for the lock among the processors, the processor stalls are eliminated with our new method.

```
C: Lock (  semaphore );
   …/*critical section*/…
   UnLock ( semaphore);
ASM: try: LL   R2,(R1)     ;read the lock
          ORI  R3,R2,1
          BEQ  R3,R2,try  ;spin if lock is busy
          SC   R3,(R1)     ;acquire the lock
          BEQ  R3,0,try   ;spin if store fails
         …/*critical section*/…
          SW   R2, (R1)    ;release lock
```

3 (a) Traditional code for spin-lock.

```
C: Lock (  semaphore );
   …/*critical section*/…
   UnLock ( semaphore);
ASM:    LW   R2,(R1)       ;read the lock
        BEQ  R2,1,sleep   ;succeed?
        …/*critical section*/…
        SW   R2,(R1)       ;release lock
        ...
sleep:  B    sleep         ;spin if lock is busy
```

3 (b) New code with our hardware support.

Figure 3. The traditional vs. new spin-lock sequence of operations in C and Assembly languages.

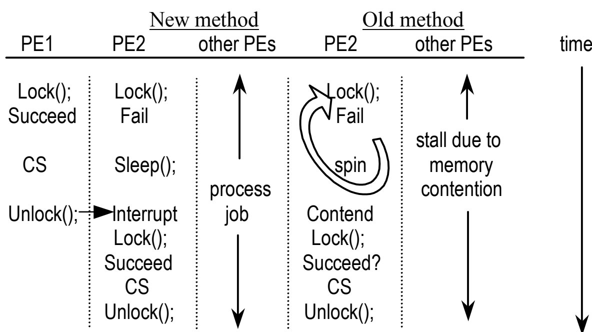

Figure 4. Comparison of the old and new methods in software flow of execution among multiple PEs in the system.

3.2. Hardware implementation

Referring back to Figure 2, and going in parallel with the above steps in software, the hardware mechanism can be explained as follows. Our SoCSU Lock unit includes lock entries mapped to an address range in the address space of each processor. For example, 256 lock entries could be mapped to the range 0xffff0000-0xffff03ff (where each lock variable is a 4-byte value). When a load instruction (LW) is executed, the incoming lock address to the decoder will enable the corresponding lock entry and the lock value residing in this entry will be put on the data bus, so the processor will read the data as if it has accessed a memory location. After this transaction, in the next clock cycle, the lock entry will be set in the SoCSU (which corresponds to ‘SC’ instruction execution in software – Figure 3 (a)). However, if the value was already a ‘1’, then this means that the processor will not be able to acquire the lock, and therefore a ‘1’ value will be put into its Pr bit location in the SoCSU indicating that the processor is waiting for the lock to be released. When the lock holder stores back a zero to the entry, i.e., releases the lock, an interrupt will be generated in the next clock cycle, enabling the next processor in line for the lock to wake up, execute its interrupt handler, and finally enter the CS.

3.3. Interrupt generation

Our hardware mechanism supports both Priority and First In First Out (FIFO) based interrupt generation. The priority assignments to the processors are re-programmable, i.e., the priorities can be modified at run time or at system reset.

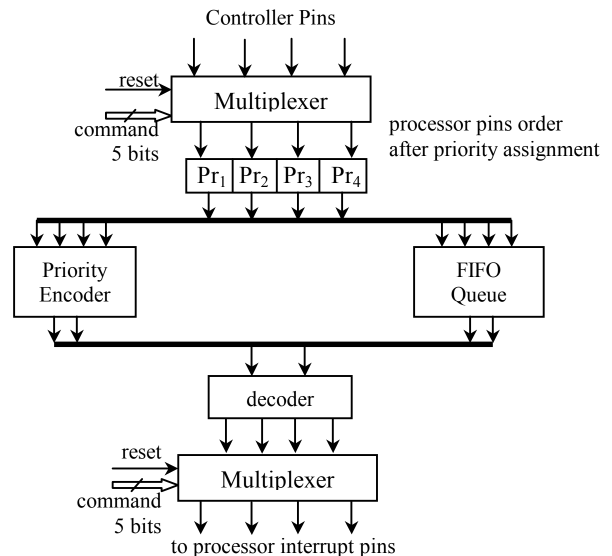

Figure 5. Priority and FIFO based interrupt generation in hardware.

As illustrated in Figure 5, the priority assignments can be programmed by the multiplexer units (at both ends) which will interpret the incoming command and establish the corresponding wire connections between the controller pins and the basic synchronization Pr bit locations and also between the decoder pins and the processors' interrupt pins. If the FIFO unit is used instead of the Priority Encoder, the priority commands do not affect the interrupt generation. Also note that the user can either use the Priority Encoder or the FIFO unit, so that he/she can enable the relevant unit whichever is preferred.

The main key feature supported by our hardware mechanism is that no matter which unit (Priority or FIFO) is used, only *one* processor is being sent a notification (after a lock release). This facility prevents unnecessary signaling to the processors in the system.

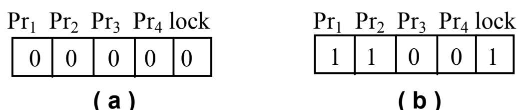

Figure 6. (a) Initial condition of lock and Pr bit locations in a four-processor system. **(b)** Bit values after PE3 acquired the lock, and PE1 and PE2 read the lock.

Example 4 Consider that we are using the Priority Encoder for interrupt generation and the priorities assigned to each processor is in descending order, i.e., PE1 has the highest priority, PE2 has the second highest priority and so on. Initially all of the Pr_i are '0' as seen in Figure 6 (a). Now, let PE3 read a lock variable at address 0xffff0000. The lock variable at 0xffff0000 in our SoCSU is set to a '1' in the following clock cycle. Just after PE3, both PE1 and PE2 also read the same lock variable (at 0xffff0000) as a '1' now (meaning that the lock is not available). Then two separate actions occur, one in hardware and the other in software. The hardware action can be explained as follows: in the SoCSU, after PE1 and PE2 read the lock as a '1', their corresponding bit locations Pr_1 and Pr_2 (associated with the lock address 0xffff0000), are set to '1' in the next half clock cycle as shown in Figure 6 (b). The Pr_1 and Pr_2 bits indicate that PE1 and PE2 are waiting for the lock variable at 0xffff0000 to be released. On the other hand, the software actions on processors PE1 and PE2 are as follows: a comparison of the lock variable with value '0', (since the value read was '1') interpreting the result of this comparison as a failure to lock the variable and therefore sleeping. After PE3 releases the lock (by storing back a '0'), the SoCSU will send an interrupt to PE1 and clear the Pr_1 bit. PE1 will therefore execute the external Interrupt Service Routine (ISR) that enables the sleeping task in PE1 to return back to its original program flow (i.e., acquire the lock and enter the CS). The ISR (Figure 7) is composed of 3 lines of assembly code, it stores back the initial value of Link Register to the SRR0 so that the processor jumps to the last line before sleeping, i.e., lock primitive execution. This will ensure the lock acquisition for PE1. After PE1 finishes its CS, it releases the lock and SoCSU sends an interrupt to PE2 enabling PE2 to acquire the lock and enter into its CS.□

```
mflr   %r0
mtspr  %SRR0, %r0
rfi
```

Figure 7. ISR assembly code for MPC750.

Tasks unable to acquire a lock should not be preempted. Otherwise, the waiting processors will not be able to forward the incoming interrupt to the correct task (which is waiting for the lock to be released). Therefore, we disable the scheduler in the RTOS before executing the locking primitive and re-enable it after acquiring the lock. This approach provides better performance for small CSes, since the context switching of tasks introduce a great overhead and it is very likely that the lock will be released before the context switching is completed.

As explained in the example above, the total number of instructions executed after the interrupt received is three (Figure 7). In other words, there is no need to save the context before the ISR execuion. Otherwise, there would be an overhead in interrupt handling and this would cause the system responsiveness to be reduced significantly (which is critical for real-time applications).

4. Experimental results

Our simulation tool is the Seamless Co-Verification Environment (Seamless CVE) [12]. For the Motorola PowerPC 750 and PowerPC 860 processors (MPC750 and MPC860), Seamless CVE provides processor model support packages together with Instruction Set Simulators (ISS) which are tightly coupled to a hardware simulator (we use Synopsys VCS™ Verilog simulator). In order to test our design, we have established the interfaces between SoCSU and MPC860 and also MPC750 RISC processors. Also, we have performed a multiprocessor simulation using four MPC750 processors each connected to the SoCSU (Figure 1) through an arbiter and a memory controller.

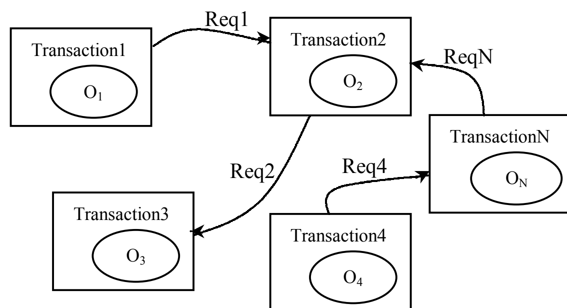

Figure 8. Database application example transactions.

As an RTOS, we have installed uC/OS-II [13] on each processor. We have run 10 tasks on top of the RTOS on each processor. In total, these 40 tasks are from a subset of a database application which constitutes a good example for thread level synchronization scenarios (Figure 8). Each thread must acquire a lock before initiating a transaction. A transaction is a process of accessing a database (labeled as O_i –objects) which is equivalent to a CS in our simulations. For example, in Figure 8, 'Req1' is the request initiated from transaction1 to acquire the lock for accessing Object2 (O_2). Other signals in the figure also refer to lock acquisition requests of the transactions.

We have combined the above database application with a client-server pair execution model for a shared memory multiprocessor system. SoCSU provides the synchronization needed between the processors that are storing and fetching information to and from the shared memory region. There exist 10 server tasks on one PE and a total of 30 client tasks on the other 3 PEs. The client-server database file-copying transactions can be explained as follows (Figure 9):

- The server gets access to a shared memory object after acquiring a lock from SoCSU.
- The server reads from its own local memory into the shared memory object.
- When the read is complete, the server notifies the client by releasing the lock.
- The client task acquires the lock and gets the data from the shared memory object to its own local memory.

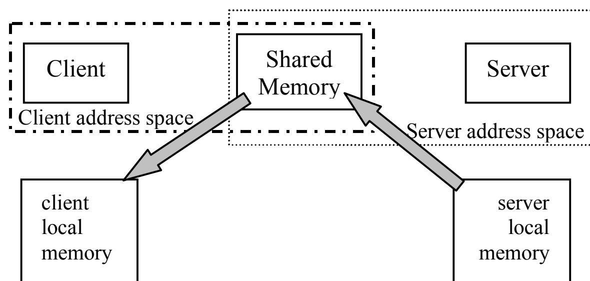

Figure 9. Copying database file from server to client.

We performed several sets of experiments. In the first set, we used traditional spin-lock primitive (test_and_set) and in the second set we used our own primitives which are using SoCSU for locking operations. The results of these two simulations are shown in Table 1.

Table 1. Worst-case experimental results for 4-MPC750 and 40 tasks simulation (comparing SoCSU approach with the traditional spin-lock method).

	Spin-Lock	SoCSU
Lock Latency (# clk cycles)	17	3.5
Max. Lock Delay (# clk cycles)	15578	34.5
Total Execution Time(#clk cycles)	1326311	1040714

For a four-processor simulation, the lock latency (Definition 2) is improved by a factor of more than 4.8, the worst-case lock delay (Definition 1) by a factor of more than 451 and the total execution time by 1.27, which indicates a 27% overall speedup in the application. The reduction in lock latency is due to the reduced assembly code size (of the locking primitive) plus the hardware support as explained in Sections 3.1 and 3.2. The reduction in lock delay is due to the fact that in our mechanism, we eliminate the spin-lock primitives, i.e., we eliminate the time spent between lock acquisition attempts in the spin-lock loops (remember the **try** labels in Figure 3(a)). Moreover, with our hardware support, irrespective of the number of processors in the system, this lock delay is a fixed number of cycles: the sums of interrupt latency (1 to 3 instruction cycles), ISR execution time and the lock latency. On the other hand, the lock delay for the traditional spin-lock approach grows exponentially as the number of processors in the system increases. Since the ping-pong effect as described in [8,9] (**try** spin-loops in Figure 3(a)), may cause continuous invalidation or update of other processors' caches. On the other hand, the cache invalidation (or update) penalty in the case of spin-on-read mechanism indicates a complexity of $O(n^2)$ in the bus traffic (where n is the total number of processors in the system) [2,9]. Similarly, it has been shown that the other approaches like barrier synchronization with counters could achieve $O(n\log n)$ proportionality [2,10,11] and the CBL scheme with private caches [2] has a bus traffic complexity of O(n) (Table 2).

Table 2. Bus traffic (contention) complexities of different mechanisms.

mechanism	Spin	Spin-on-read	CBL	Barrier (counters)	SoCSU
Traffic	Exponential	$O(n^2)$	O(n)	O(nlogn)	constant

Therefore, for all of these approaches, the lock delay is scaled with the number of processors in the system (since the lock delay is dependent on the bus contention complexities as listed in Table 2).

However, in our approach, there is neither contention for the lock nor any penalty due to cache invalidation. This is because the processors do not spin either in the memory or in the caches, but instead wait for an interrupt. Therefore, in case of a lock release, the notified processor executes the ISR before acquiring the lock.

5. Conclusion

In this paper, we presented a hardware mechanism (SoCSU) which handles processor synchronization removing software overhead and therefore improving delay, bandwidth and latency. SoCSU eliminates the intervention of the main memory bus, hence enabling the memory bus to be used for other useful work. Furthermore, SoCSU provides a notification mechanism via interrupt generation to the processors that are in queue to acquire the lock. The SoCSU brings a reduction in software code size and also enables the elimination of special load and store instructions.

Our hardware mechanism imposes no memory or cache-consistency overheads, it allows each processor to try to acquire a lock when they have a *chance of getting the lock*. Our solution simplifies software and allows the system to minimize spin-costs. The simulation results show 450% speedup in the lock delay, 380% speedup in lock latency and 27% overall speedup in total execution time of a database application example in case of a four-processor system. Furthermore, as we increase the number of processors and CS execution frequency, this would scale our performance improvement significantly.

For our future work, we intend to extend our approach to handle long critical sections. Therefore, we plan to add mechanisms to enable preemption of sleeping tasks which are waiting for lock(s) to become freed.

6. Acknowledgements

We wish to thank Jaehwan Lee for his help with the simulation of the client-server example.

This research is funded by the State of Georgia under the Yamacraw Initiative and by NSF under INT-9973120, CCR-9984808 and CCR-0082164.

We also acknowledge software donations from Mentor Graphics and Synopsys as well as hardware donations from Sun and Intel.

7. References

[1] T.E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors", IEEE Trans. Parallel Distrb. Syst. **1**, no 1, Jan. 1990, pp. 6-16.
[2] U. Ramachandran, J. Lee, "Cache-based synchronization in shared memory multiprocessors", Journal of Parallel and Ditrb. Computing. **32**, 1996, pp. 11-27.
[3] U. Ramachandran, J. Lee, "Processor initiated sharing in multiprocessor caches", Tech. Rep. GIT-ICS-88/43, Georgia Institute of Technology, Nov. 1988.
[4] J.R. Goodman, M.K. Vernon, and P.J. Woest, "Efficient Synchronization Primitives for large-scale cache-coherent multiprocessors", Proc. of the Third International Conference on ASPLOS, April 1989, pp. 64-75.
[5] G. Graunke, S. Thakkar, "Synchronization algorithms for shared-memory multiprocesors", IEEE Computer **C-23** (6), 1990, pp. 60-69.
[6] U. Ramachandran, J. Lee, "Architectural primitives for a scalable memory multiprocessor", Tech. Rep. GIT-ICS-91/10, Georgia Institute of Technology, Feb. 1991.
[7] Joe Heinrich, "MIPS R4000 Microprocessor User's Manual", *2nd* edition, pp. 286-291.
[8] M. Dubois, C. Scheurich, F.A. Briggs, "Synchronization, coherence, and event ordering in multiprocessors", IEEE Computer, February 1988, pp. 9-21.
[9] U. Ramachandran, J. Lee, "Synchronization with Multiprocessor Caches", Tech. Rep. GIT-ICS-90-15, Georgia Institute of Technology, March 1990.
[10] D. Hensgen, R. Finkel, U. Manber, "Two algorithms for barrier synchronization", Internat. J. Parallel Programming, **17**, 1, February 1988, pp 1-17.
[11] J. M. Mellor-Crummey, M.L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors", ACM Trans. Comput. Syst., **9**, 1, February 1991, pp 21-65.
[12] Mentor Graphics, Hardware/Software Co-Verification: Seamless, http://www.mentor.com/seamless/.
[13] Labrosse, Jean J., MicroC/OS-II The Real-Time Kernel, R&D Books, Miller Freeman, Inc., Lawrence, KS, 1999.